

This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representation of
The original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images,
please do not report the images to the
Image Problem Mailbox.**

THIS PAGE BLANK (USPTO)



15 SEP 2000

P.D. 03 OCT 2000

WIPO

PCT

BREVET D'INVENTION

CERTIFICAT D'UTILITÉ - CERTIFICAT D'ADDITION**COPIE OFFICIELLE**

Le Directeur général de l'Institut national de la propriété industrielle certifie que le document ci-annexé est la copie certifiée conforme d'une demande de titre de propriété industrielle déposée à l'Institut.

Fait à Paris, le **22 AOUT 2000**

Pour le Directeur général de l'Institut
national de la propriété industrielle
Le Chef du Département des brevets

**PRIORITY
DOCUMENT**

SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH RULE 17.1(a) OR (b)

Martine PLANCHE

INSTITUT
NATIONAL DE
LA PROPRIÉTÉ
INDUSTRIELLE

SIEGE

26 bis, rue de Saint Petersburg
75800 PARIS Cédex 08
Téléphone : 01 53 04 53 04
Télécopie : 01 42 93 59 30

THIS PAGE BLANK (USPTO)

REQUÊTE EN DÉLIVRANCE

26 bis, rue de Saint Pétersbourg
75800 Paris Cedex 08
Téléphone : 01 53 04 53 04 Télécopie : 01 42 93 59 30

Confirmation d'un dépôt par télécopie ☐

Cet imprimé est à remplir à l'encre noire en lettres capitales

Réservé à l'INPI

DATE DE REMISE DES PIÈCES **23 AOUT 1 9**
N° D'ENREGISTREMENT NATIONAL **9910697**
DÉPARTEMENT DE DÉPÔT **75 INPI PARIS**
DATE DE DÉPÔT **23 AOUT 1999**

1 **NOM ET ADRESSE DU DEMANDEUR OU DU MANDATAIRE
À QUI LA CORRESPONDANCE DOIT ÊTRE ADRESSÉE**

**CABINET PLASSERAUD
84, RUE D'AMSTERDAM
F-75440 PARIS CEDEX 09**

2 **DEMANDE** Nature du titre de propriété industrielle

☒ brevet d'invention ☐ demande divisionnaire
☐ certificat d'utilité ☐ transformation d'une demande
de brevet européen

☐ demande initiale
☐ brevet d'invention

n° du pouvoir permanent / références du correspondant **MF-BFF990196** téléphone **01 44 63 4111**

Établissement du rapport de recherche

☐ différé ☒ immédiat

Le demandeur, personne physique, requiert le paiement échelonné de la redevance ☐ oui ☐ non

Titre de l'invention (200 caractères maximum)

**PROTOCOLE DE GESTION, PROCEDE DE VERIFICATION ET DE TRANSFORMATION
D'UN FRAGMENT DE PROGRAMME TELECHARGE ET SYSTEMES CORRESPONDANTS.**

3 **DEMANDEUR (S)** n° SIREN

code APE-NAF

Nom et prénoms (souligner le nom patronymique) ou dénomination

TRUSTED LOGIC

Forme juridique

Société anonyme

Nationalité (s) **française**

Adresse (s) complète (s)

Pays

**23, avenue de Fulpmes
78450 VILLEPREUX**

FRANCE

4 **INVENTEUR (S)** Les inventeurs sont les demandeurs ☐ oui ☒ non

En cas d'insuffisance de place, poursuivre sur papier libre ☐

Si la réponse est non, fournir une désignation séparée

5 **RÉDUCTION DU TAUX DES REDEVANCES**

☐ requise pour la 1ère fois ☐ requise antérieurement au dépôt : joindre copie de la décision d'admission

6 **DÉCLARATION DE PRIORITÉ OU REQUÊTE DU BÉNÉFICE DE LA DATE DE DÉPÔT D'UNE DEMANDE ANTÉRIEURE**

pays d'origine

numéro

date de dépôt

nature de la demande

7 **DIVISIONS**

antérieures à la présente demande n°

date

n°

date

8 **SIGNATURE DU DEMANDEUR OU DU MANDATAIRE**

(nom et qualité du signataire)

SIGNATURE DU PRÉPOSÉ À LA RÉCEPTION

SIGNATURE APRÈS ENREGISTREMENT DE LA DEMANDE À L'INPI

J. FORT
CPI Brevets n° 92-1092

X

Elle garantit un droit d'accès et de rectification pour les données vous concernant auprès de l'INPI.



MF/MHM - BFF990196

BREVET D'INVENTION, CERTIFICAT D'UTILITE

DÉSIGNATION DE L'INVENTEUR

(si le demandeur n'est pas l'inventeur ou l'unique inventeur)

DEPARTEMENT DES BREVETS

26bis, rue de Saint-Petersbourg
75800 Paris Cédex 08

Tél. : 01 53 04 53 04 - Télécopie : 01 42 93 59 30

N° D'ENREGISTREMENT NATIONAL

99-10697

TITRE DE L'INVENTION: Protocole de gestion, procédé de vérification et de transformation d'un fragment de programme téléchargé et systèmes correspondants

La société titulaire : TRUSTED LOGIC
ayant pour mandataire :

LE(S) SOUSSIGNÉ(S)
CABINET PLASSERAUD
84, rue d'Amsterdam
75440 PARIS CEDEX 09

DÉSIGNE(NT) EN TANT QU'INVENTEUR(S) (indiquer nom, prénoms, adresse et souligner le nom patronymique) :

LEROY Xavier
88, avenue de Paris
78000 VERSAILLES - FRANCE

NOTA : A titre exceptionnel, le nom de l'inventeur peut être suivi de celui de la société à laquelle il appartient (société d'appartenance) lorsque celle-ci est différente de la société déposante ou titulaire.

Date et signature (s) du (des) demandeur (s) ou du mandataire

20 août 1999

J. FORT
CPI Brevets n° 92-1092

PROTOCOLE DE GESTION, PROCEDE DE VERIFICATION ET DE
TRANSFORMATION D'UN FRAGMENT DE PROGRAMME TELECHARGE ET
SYSTEMES CORRESPONDANTS

5 L'invention concerne un protocole de gestion, un
procédé de vérification, un procédé de transformation d'un
fragment de programme téléchargé et les systèmes
correspondants, plus particulièrement destinés aux
systèmes informatiques embarqués disposant de faibles
10 ressources en mémoire et en puissance de calcul.

D'une manière générale, en référence à la figure
1a, on rappelle que les systèmes informatiques embarqués
10 comportent un microprocesseur 11, une mémoire
permanente, telle qu'une mémoire non inscriptible 12
15 contenant le code du programme exécutable, une mémoire
permanente, non volatile, réinscriptible 13 de type EEPROM
contenant les données stockées dans le système, une
mémoire vive, volatile, 14 dans laquelle le programme
stocke ses résultats intermédiaires pendant son exécution,
20 et des dispositifs d'entrée/sortie 15 permettant au
système d'interagir avec son environnement. Dans le cas où
le système informatique embarqué est constitué par une
carte à microprocesseur, du type carte bancaire, le
dispositif d'entrée/sortie 15 consiste en une liaison
25 série permettant à la carte de communiquer avec un
terminal, tel qu'un terminal lecteur de cartes.

Dans les systèmes informatiques embarqués
classiques, le code du programme exécuté par le système
est figé lors de la construction du système, ou, au plus
30 tard, lors de la personnalisation de ce dernier avant
livraison à l'utilisateur final.

Des systèmes informatiques embarqués plus évolués ont toutefois été mis en œuvre, ces systèmes étant reprogrammables, comme par exemple les cartes à microprocesseur de type JavaCard. Ces systèmes reprogrammables ajoutent, vis-à-vis des précédents, la possibilité d'enrichir le programme après la mise en service du système, par une opération de téléchargement de fragments de programmes. Ces fragments de programmes, communément désignés par "applets" en langage anglo-saxon, seront indifféremment désignés par appliquettes ou fragments de programmes dans la présente description. Pour une description plus détaillée des systèmes JavaCard, on pourra utilement se reporter à la documentation éditée par la société SUN MICROSYSTEMS INC. Et en particulier à la documentation disponible électroniquement, chapitre JavaCard technology sur le site W.W.W. (*World Wide Web*) <http://java.sun.com/products/javacard/index.html>, juin 1999.

La figure 1b illustre l'architecture d'un tel système informatique embarqué reprogrammable. Cette architecture est semblable à celle d'un système embarqué classique, à la différence que le système embarqué reprogrammable peut en outre recevoir des appliquettes par l'intermédiaire d'un de ses dispositifs d'entrée/sortie, puis stocker ces dernières dans sa mémoire permanente 13 à partir de laquelle elles peuvent ensuite être exécutées en complément du programme principal.

Pour des raisons de portabilité entre différents systèmes informatiques embarqués, les appliquettes se présentent sous forme de code pour une machine virtuelle standard. Ce code n'est pas directement exécutable par le

microprocesseur 11 mais doit être interprété de manière
logicielle par une machine virtuelle 16, laquelle est
constituée par un programme résidant en mémoire permanente
non inscriptible 12. Dans l'exemple précité des cartes
5 JavaCard, la machine virtuelle utilisée est un sous-
ensemble de la machine virtuelle Java. Pour une
description des spécifications relatives à la machine
virtuelle Java et de la machine virtuelle utilisée, on
pourra utilement se reporter à l'ouvrage publié par Tim
10 LINDHOLM et Frank YELLIN, intitulé "*The Java Virtual
Machine Specification*", Addison-Wesley 1996, et à la
documentation éditée par la société SUN MICROSYSTEMS INC.
"*JavaCard 2.1 Virtual Machine Specification*",
documentation disponible électroniquement sur le site
15 W.W.W. <http://java.sun.com/products/javacard/JCVMSpec.pdf>,
mars 1999.

L'opération de téléchargement d'appliquettes sur
un système informatique embarqué en service pose
d'importants problèmes de sécurité. Une appliquette
20 involontairement, voire volontairement, mal écrite peut
modifier de manière incorrecte des données présentes sur
le système, empêcher le programme principal de s'exécuter
correctement ou en temps voulu, ou encore modifier
d'autres appliquettes téléchargées antérieurement, en
25 rendant celles-ci inutilisables ou nuisibles.

Une appliquette écrite par un pirate informatique
peut également divulguer des informations confidentielles
stockées dans le système, informations telles que le code
d'accès dans le cas d'une carte bancaire par exemple.

A l'heure actuelle, trois solutions ont été proposées en vue de remédier au problème de sécurité des appli-quettes.

5 Une première solution consiste à utiliser des signatures cryptographiques afin de n'accepter que des appli-quettes provenant de personnes ou d'organismes de confiance.

Dans l'exemple d'une carte bancaire précité, seules les appli-quettes portant la signature cryptographique de la
10 banque ayant émis la carte sont acceptées et exécutées par la carte, toute autre appli-quette non signée étant rejetée au cours de l'opération de téléchargement. Un utilisateur mal intentionné de la carte, ne disposant pas de clés de chiffrement de la banque, sera donc dans l'incapacité de
15 faire exécuter une appli-quette non signée et dangereuse sur la carte.

Cette première solution est bien adaptée au cas où toutes les appli-quettes proviennent d'une même source unique, la banque dans l'exemple précité. Cette solution est
20 difficilement applicable au cas où les appli-quettes proviennent de plusieurs sources, comme, dans l'exemple d'une carte bancaire, le fabricant de la carte, la banque, les organismes gestionnaires des services par carte bancaire, les grands organismes de distribution
25 commerciale offrant à la clientèle des programmes de fidélisation et proposant, légitimement, de télécharger des appli-quettes spécifiques sur la carte. Le partage et la détention entre ces divers acteurs économiques des clés de chiffrement nécessaires à la signature électronique des
30 appli-quettes posent des problèmes techniques, économiques et juridiques majeurs.

Une deuxième solution consiste à effectuer des contrôles dynamiques d'accès et de typage pendant l'exécution des appliquettes.

Dans cette solution, la machine virtuelle lors de l'exécution des appliquettes, effectue un certain nombre de contrôles, tels que :

- contrôle d'accès à la mémoire : à chaque lecture ou écriture d'une zone mémoire, la machine virtuelle vérifie le droit d'accès de l'appliquette aux données correspondantes ;
- vérification dynamique des types de données : à chaque instruction de l'appliquette, la machine virtuelle vérifie que les contraintes sur les types de données sont satisfaites. A titre d'exemple, la machine virtuelle peut traiter spécialement les données telles que des adresses mémoire valides et empêcher que l'appliquette n'engendre des adresses mémoire invalides par l'intermédiaire de conversions entier/adresse ou d'opérations arithmétiques sur les adresses ;
- détection des débordements de pile et des accès illégaux dans la pile d'exécution de la machine virtuelle, lesquels, dans certaines conditions, sont susceptibles de perturber le fonctionnement de cette dernière, au point de contourner les mécanismes de contrôle précédents.

Cette deuxième solution permet l'exécution d'une large gamme d'appliquettes dans des conditions de sécurité satisfaisantes. Elle présente toutefois l'inconvénient d'un ralentissement considérable de l'exécution, provoqué par l'ensemble des vérifications dynamiques. Pour obtenir une réduction de ce ralentissement, une partie de ces

vérifications peut être prise en charge par le microprocesseur lui-même, au prix toutefois d'une augmentation de la complexité de ce dernier et donc du prix de revient du système embarqué. De telles
 5 vérifications augmentent en outre les besoins en mémoire vive et permanente du système, en raison des informations supplémentaires de type qu'il est nécessaire d'associer aux données manipulées.

Une troisième solution consiste à effectuer une
 10 vérification statique du code de l'appliquette lors du téléchargement.

Dans cette solution, cette vérification statique simule l'exécution de l'appliquette au niveau des types de données et établit, une fois pour toutes, que le code de
 15 l'appliquette respecte la règle des types de données et de contrôle d'accès imposée par la machine virtuelle et ne provoque pas de débordement de pile. Si cette vérification statique réussit, l'appliquette peut alors être exécuté sans qu'il soit nécessaire de vérifier dynamiquement que

20 cette règle est respectée. Dans le cas où le processus de vérification statique échoue, le système embarqué rejette l'"appliquette" et ne permet pas son exécution ultérieure. Pour une description plus détaillée de la troisième solution précitée, on pourra utilement se reporter à
 25 l'ouvrage édité par Tim LINDHOLM et Frank YELLIN précédemment cité, à l'article publié par James A.GOSLING intitulé "*Java Intermediate Byte Codes*", Actes du ACM SIGPLAN, Workshop on Intermediate Representations (IR'95), pages 111-118, janvier 1995, et au brevet US 5,748,964
 30 délivré le 05/05/1998.

Vis-à-vis de la deuxième solution, la troisième solution présente l'avantage d'une exécution des appli-
 quettes beaucoup plus rapide, puisque la machine virtuelle n'effectue aucune vérification pendant
 5 l'exécution.

La troisième solution présente toutefois l'inconvénient d'un processus de vérification statique du code complexe et coûteux, tant en taille de code nécessaire pour conduire ce processus, qu'en taille de mémoire vive
 10 nécessaire pour contenir les résultats intermédiaires de la vérification, et qu'en temps de calcul. A titre d'exemple illustratif, la vérification de code intégré dans le système Java JDK commercialisé par SUN MICROSYSTEMS représente de l'ordre de 50 k-octets de code
 15 machine, et sa consommation en mémoire vive est proportionnelle à $(T_p + T_r) \times N_b$ où T_p désigne l'espace maximum de pile, T_r désigne le nombre maximum de registres et N_b désigne le nombre maximum de cibles de branchements utilisées par un sous-programme, encore communément

20 désigné par méthode, de l'appliquette Ces besoins en mémoire dépassent largement les capacités des ressources de la majorité des systèmes informatiques embarqués actuels, notamment des cartes à microprocesseur commercialement disponibles.

25 Plusieurs variantes de la troisième solution ont été proposées, dans lesquelles l'éditeur de l'appliquette transmet au vérificateur, outre le code de l'appliquette,
 30 un certain nombre d'informations supplémentaires spécifiques telles que types de données précalculés ou preuve préétablie de typage de données correct. Pour une description plus détaillée des modes opératoires

correspondants, on pourra utilement se reporter aux articles publiés par Eva ROSE et Kristoffer HØGSBRO ROSE, *"Lightweight Bytecode Verification"*, Actes du Workshop Formal Underspinning of Java, Octobre 1998, et par George C. NECULA, *"Proof-Carrying Code"*, Actes du 24^e ACM Symposium Principles of Programming Languages, pages 106-119 respectivement.

Ces informations supplémentaires permettent de vérifier le code plus rapidement et de réduire légèrement la taille du code du programme de vérification mais ne permettent toutefois pas de réduire les besoins en mémoire vive, voire les augmentent, de manière très importante, dans le cas des informations de preuve préétablie de typage de données correct.

La présente invention a pour objet de remédier aux inconvénients précités de l'art antérieur.

En particulier, un objet de la présente invention est la mise en œuvre d'un protocole de gestion d'un fragment de programme, ou appliquette, téléchargé

permettant une exécution sur de ce dernier par un système informatique embarqué disposant de faibles ressources, tel qu'une carte à microprocesseur.

Un autre objet de la présente invention est également la mise en œuvre d'un procédé de vérification d'un fragment de programme, ou appliquette, téléchargé dans lequel un processus de vérification statique du code de l'appliquette est conduit lors de son téléchargement, ce processus pouvant être rapproché, au moins dans son principe, de la troisième solution précédemment décrite, mais dans lequel des techniques nouvelles de vérification sont introduites, afin de permettre l'exécution de cette

vérification dans les limites de valeurs de taille mémoire et de vitesse de calcul imposées par les cartes à microprocesseur et autres systèmes informatiques embarqués peu puissants.

5 Un autre objet de la présente invention est également la mise en œuvre de procédés de transformation de fragments de programmes de type classique obtenus par exemple par la mise en œuvre d'un compilateur Java en fragments de programmes, ou appliquettes, normalisés, 10 satisfaisant a priori aux critères de vérification du procédé de vérification objet de l'invention, en vue d'accélérer le processus de vérification et d'exécution de ces derniers au niveau des systèmes informatiques embarqués ou cartes à microprocesseur actuels.

15 Un autre objet de la présente invention est, enfin, la réalisation de systèmes informatiques embarqués permettant la mise en œuvre du protocole de gestion et du procédé de vérification d'un fragment de programme téléchargé précités ainsi que de systèmes informatiques 20 permettant la mise en œuvre des procédés de transformation de fragments de programmes, ou appliquettes, classiques en fragments de programmes, ou appliquettes, normalisés précités.

25 Le protocole de gestion d'un fragment de programme téléchargé sur un système embarqué reprogrammable, objet de la présente invention, s'applique notamment à une carte à microprocesseur munie d'une mémoire réinscriptible. Le ~~fragment de programme est constitué par un code objet,~~ suite d'instructions, exécutable par le microprocesseur du 30 système embarqué par l'intermédiaire d'une machine virtuelle munie d'une pile d'exécution et de registres ou

variables locales manipulées par ces instructions et permettant d'interpréter ce code objet. Le système embarqué est interconnecté à un terminal.

Il est remarquable en ce qu'il consiste au moins, au
5 niveau du système embarqué, à détecter une commande de téléchargement du fragment de programme. Sur réponse positive à l'étape consistant à détecter une commande de téléchargement, il consiste en outre à lire le code objet constitutif du fragment de programme et à mémoriser
10 temporairement ce code objet dans la mémoire réinscriptible. L'ensemble du code objet mémorisé est soumis à un processus de vérification instruction par instruction. Le processus de vérification consiste au moins en une étape d'initialisation de la pile des types
15 et du tableau des types de registres représentant l'état de la machine virtuelle au début de l'exécution du code objet mémorisé temporairement et en une succession d'étapes de vérification instruction par instruction de l'existence, pour chaque instruction courante, d'une
20 cible, cible d'instruction de branchement, cible d'un gestionnaire d'exceptions, et en une vérification et une actualisation de l'effet de l'instruction courante sur la pile des types et sur le tableau des types de registres. Dans le cas d'une vérification non réussie du code objet,
25 le protocole objet de l'invention consiste à effacer le fragment de programme enregistré momentanément, en l'absence d'enregistrement de ce dernier dans le répertoire de fragments de programmes disponibles, et à adresser au lecteur un code d'erreur.

30 Le procédé de vérification d'un fragment de programme téléchargé sur un système embarqué, objet de

l'invention, s'applique notamment à une carte à microprocesseur munie d'une mémoire réinscriptible. Le fragment de programme est constitué par un code objet et comporte au moins un sous-programme, suite d'instructions, exécutable par le microprocesseur du système embarqué par l'intermédiaire d'une machine virtuelle munie d'une pile d'exécution et de registres d'opérandes manipulés par ces instructions et permettant d'interpréter ce code objet. Le système embarqué est interconnecté à un lecteur.

Il est remarquable en ce que, suite à la détection d'une commande de téléchargement et à la mémorisation du code objet constitutif du fragment de programme dans la mémoire réinscriptible, il consiste, pour chaque sous-programme, à effectuer une étape d'initialisation de la pile des types et du tableau des types de registres par des données représentant l'état de la machine virtuelle au début de l'exécution du code objet mémorisé temporairement, à effectuer une vérification du code objet mémorisé temporairement instruction par instruction, par

discrimination de l'existence, pour chaque instruction courante, d'une cible d'instruction de branchement, d'une cible d'un appel d'un gestionnaire d'exceptions ou d'une cible d'un appel de sous-routine et à effectuer une vérification et une actualisation de l'effet de l'instruction courante sur les types de données de la pile des types et du tableau des types de registres, en fonction de l'existence d'une cible d'instruction de branchement, d'une cible d'un appel de sous-routine ou d'une cible d'un appel de gestionnaire d'exceptions. La vérification est réussie lorsque le tableau des types de registres n'est pas modifié au cours d'une vérification de

toutes les instructions, le processus de vérification étant poursuivi instruction par instruction jusqu'à ce que le tableau des types de registres soit stable, en l'absence de modification. Le processus de vérification est interrompu sinon.

Le procédé de transformation d'un code objet d'un fragment de programme en un code objet normalisé pour ce même fragment de programme, objet de la présente invention, s'applique à un code objet d'un fragment de programme dans lequel les opérandes de chaque instruction appartiennent aux types de données manipulées par cette instruction, la pile d'exécution ne présente pas de phénomène de débordement et pour chaque instruction de branchement le type des variables de la pile au niveau de ce branchement est le même qu'au niveau des cibles de ce branchement. Le code objet normalisé obtenu est tel que les opérandes de chaque instruction appartiennent aux types de données manipulées par cette instruction, la pile d'exécution ne présente pas de phénomène de débordement et

la pile d'exécution est vide à chaque instruction de cible de branchement.

Il est remarquable en ce qu'il consiste, pour l'ensemble des instructions du code objet, à annoter chaque instruction courante par le type de données de la pile d'exécution avant et après l'exécution de cette instruction, les données d'annotation étant calculées au moyen d'une analyse du flot des données relatif à cette instruction, à détecter au sein des instructions, et de chaque instruction courante, l'existence de branchements pour lesquels la pile d'exécution n'est pas vide, l'opération de détection étant conduite à partir des

données d'annotation du type de variables de pile allouées à chaque instruction courante. En présence d'une détection d'une pile d'exécution non vide, il consiste en outre à insérer des instructions de transfert des variables de pile de part et d'autre de ces branchements ou de ces cibles de branchement afin de vider le contenu de la pile d'exécution dans des registres temporaires avant ce branchement et de rétablir la pile d'exécution à partir des registres temporaires après ce branchement et à n'insérer aucune instruction de transfert sinon.

Ce procédé permet ainsi d'obtenir un code objet normalisé pour ce même fragment de programme, dans lequel la pile d'exécution est vide à chaque instruction de branchement et de cible de branchement, en l'absence de toute modification de l'exécution du fragment de programme.

Le procédé de transformation d'un code objet d'un fragment de programme en un code objet normalisé pour ce même fragment de programme, objet de la présente invention, s'applique, en outre, à un code objet d'un fragment de programme dans lequel les opérandes de chaque instruction appartiennent aux types de données manipulées par cette instruction, et un opérande de type déterminé écrit dans un registre par une instruction de ce code objet est relu depuis ce même registre par une autre instruction de ce code objet avec le même type de donnée déterminé. Le code objet normalisé obtenu est tel que les opérandes appartiennent aux types de données manipulées par cette instruction, un seul et même type de donnée étant alloué à un même registre dans tout le code objet normalisé.

Il est remarquable en ce qu'il consiste, pour l'ensemble des instructions du code objet, à annoter chaque instruction courante par le type de donnée des registres avant et après l'exécution de cette instruction, les données d'annotation étant calculées au moyen d'une analyse du flot des données relatif à cette instruction, et à effectuer une réallocation des registres d'origine employés avec des types différents, par division de ces registres d'origine en registres normalisés distincts. Un registre normalisé est alloué à chaque type de donnée utilisé. Une réactualisation des instructions qui manipulent les opérandes qui font appel aux registres normalisés est effectuée.

Le protocole de gestion d'un fragment de programme, le procédé de vérification d'un fragment de programme, les procédés de transformation de code objet de fragments de programmes en code objet normalisé et les systèmes correspondants, objets de la présente invention, trouvent application au développement des systèmes embarqués reprogrammables, tels que les cartes à microprocesseur, notamment dans l'environnement Java.

Il seront mieux compris à la lecture de la description et à l'observation des dessins ci-après, dans lesquels, outre les figures 1a et 1b relatives à l'art antérieur :

- la figure 2 représente un organigramme illustratif du protocole de gestion d'un fragment de programme téléchargé sur un système embarqué reprogrammable,
- la figure 3a représente, à titre illustratif, un organigramme d'un procédé de vérification d'un fragment

de programme téléchargé conformément à l'objet de la présente invention,

- 5 - la figure 3b représente un diagramme illustratif des types de données et des relations de sous-typage mis en œuvre par le procédé de gestion et le procédé de vérification d'un fragment de programme téléchargé, objet de la présente invention,
 - 10 - la figure 3c représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion d'une instruction de branchement,
 - 15 - la figure 3d représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion d'une instruction d'appel de sous-routine,
 - la figure 3e représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion d'une cible d'un gestionnaire d'exceptions,
 - la figure 3f représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion d'une cible de branchements incompatibles,

 - 20 - la figure 3g représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion d'une absence de cible de branchement,
 - la figure 3h représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion de l'effet de l'instruction courante sur la pile des types,
 - 25 - la figure 3i représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion d'une instruction de lecture d'un registre,
-

- la figure 3j représente un détail du procédé de vérification selon la figure 3a, relatif à la gestion d'une instruction d'écriture d'un registre,
- 5 - la figure 4a représente un organigramme illustratif d'un procédé de transformation d'un code objet d'un fragment de programme en un code objet normalisé pour ce même fragment de programme à instruction de branchement respectivement de cible de branchement à pile vide,
- 10 - la figure 4b représente un organigramme illustratif d'un procédé de transformation d'un code objet d'un fragment de programme en un code objet normalisé pour ce même fragment de programme faisant appel à des registres typés, à chaque registre étant attribué un
- 15 seul type de donnée spécifique,
- la figure 5a représente un détail de mise en œuvre du procédé de transformation illustré en figure 4a,
- la figure 5b représente un détail de mise en œuvre du ~~procédé de transformation illustré en figure 4b,~~
- 20 - la figure 6 représente un schéma fonctionnel de l'architecture complète d'un système de développement d'un fragment de programme normalisé, et d'une carte à microprocesseur reprogrammable permettant la mise en œuvre du protocole de gestion et du procédé de
- 25 vérification d'un fragment de programme conformément à l'objet de la présente invention.

D'une manière générale, on indique que le

protocole de gestion, le procédé de vérification et de transformation d'un fragment de programme téléchargé,

30 objet de la présente invention, ainsi que les systèmes correspondants, sont mis en œuvre grâce à une architecture

logicielle pour le téléchargement et l'exécution sûrs d'appliquettes sur un système informatique embarqué disposant de faibles ressources, tel que notamment les cartes à microprocesseur.

5 D'une manière générale, on indique que la description ci-après concerne l'application de l'invention dans le contexte des cartes à microprocesseur reprogrammables de type JavaCard, confer documentation disponible électroniquement auprès de la société SUN
10 MICROSYSTEMS INC. rubrique JavaCard Technology précédemment mentionnée dans la description.

Toutefois, la présente invention est applicable à tout système embarqué reprogrammable par l'intermédiaire d'un téléchargement d'appliquette écrite dans le code
15 d'une machine virtuelle comprenant une pile d'exécution, des registres ou variables locales et dont le modèle d'exécution est fortement typé, chaque instruction du code de l'appliquette ne s'appliquant qu'à des types de données spécifiques. Le protocole de gestion d'un fragment de
20 programme téléchargé sur un système embarqué reprogrammable, objet de la présente invention, sera maintenant décrit de manière plus détaillée en liaison avec la figure 2.

En liaison avec la figure précitée, on indique que
25 le code objet constitutif du fragment de programme ou appliquette est constitué par une suite d'instructions exécutables par le microprocesseur du système embarqué par l'intermédiaire de la machine virtuelle précédemment
mentionnée. La machine virtuelle permet d'interpréter le
30 code objet précité. Le système embarqué est interconnecté

à un terminal par l'intermédiaire d'une liaison série par exemple.

En référence à la figure 2 précédemment mentionnée, le protocole de gestion objet de la présente invention consiste au moins, au niveau du système embarqué, à détecter en une étape 100a, 100b, une commande de téléchargement de ce fragment de programme. Ainsi, l'étape 100a peut consister en une étape de lecture de la commande précitée et l'étape 100b en une étape de test de la commande lue et de vérification de l'existence d'une commande de téléchargement.

Sur réponse positive à l'étape de détection 100a, 100b précitée d'une commande de téléchargement, le protocole objet de la présente invention consiste ensuite à lire à l'étape 101 le code objet constitutif du fragment de programme considéré et à mémoriser temporairement le code objet précité dans la mémoire réinscriptible du système informatique embarqué. L'étape de lecture du code objet et de mémorisation temporaire de ce dernier dans la mémoire réinscriptible est désignée par chargement du code de l'appliquette sur la figure 2.

L'étape précitée est alors suivie d'une étape 102 consistant à soumettre l'ensemble du code objet mémorisé temporairement à un processus de vérification, instruction par instruction, du code objet précité.

Le processus de vérification consiste au moins en une étape d'initialisation de la pile des types et du tableau des types de données représentant l'état de la machine virtuelle au début de l'exécution du code objet mémorisé temporairement, ainsi qu'en une succession d'étapes de vérification instruction par instruction par

discrimination de l'existence, pour chaque instruction courante, notée I_i , d'une cible telle qu'une cible d'instruction de branchement notée CIB, une cible d'appel d'un gestionnaire d'exceptions ou une cible d'un appel de sous-routine. Une vérification et une actualisation de l'effet de l'instruction courante I_i sur la pile des types et sur le tableau des types de registres est effectuée.

Lorsque la vérification est réussie à l'étape 103a, le protocole objet de la présente invention consiste à enregistrer à l'étape 104 le fragment de programme téléchargé dans un répertoire de fragments de programmes disponibles et à envoyer à l'étape 105 au lecteur un accusé de réception positif.

Au contraire, dans le cas d'une vérification non réussie du code objet à l'étape 103b, le protocole objet de l'invention consiste à effacer à l'étape 106 le fragment de programme enregistré momentanément en l'absence d'enregistrement de ce fragment de programme dans le répertoire de fragments de programmes disponibles

et, à une étape 107, à adresser au lecteur un code d'erreur. Les étapes 107 et 105 peuvent être réalisées soit séquentiellement après les étapes 106 respectivement 104, soit en opération multitâche avec celles-ci.

En référence à la même figure 2, sur réponse négative à l'étape consistant à détecter une commande de téléchargement à l'étape 100b, le protocole objet de la présente invention consiste à détecter, en une étape 108, une commande de sélection d'un fragment de programme disponible dans un répertoire de fragments de programmes et, sur réponse positive à l'étape 108, la sélection d'un fragment de programme disponible étant détectée, à appeler

à l'étape 109 ce fragment de programme disponible sélectionné en vue de son exécution. L'étape 109 est alors suivie d'une étape d'exécution 110 du fragment de programme disponible appelé par l'intermédiaire de la machine virtuelle en l'absence de toute vérification dynamique de types de variables, des droits d'accès aux objets manipulés par le fragment de programme disponible appelé ou du débordement de la pile d'exécution lors de l'exécution de chaque instruction.

Dans le cas où une réponse négative est obtenue à l'étape 108, cette étape consistant à détecter une commande de sélection d'un fragment de programme disponible appelé, le protocole objet de la présente invention consiste à procéder, à une étape 111, au traitement des commandes standard du système embarqué.

En ce qui concerne l'absence de vérification dynamique de type ou de droit d'accès aux objets de type JavaCard par exemple, on indique que cette absence de vérification ne compromet pas la sûreté de la carte car le

code de l'appliquette a nécessairement passé avec succès la vérification.

D'une manière plus spécifique, on indique que la vérification de code effectuée, conformément au procédé objet de l'invention, sur la carte à microprocesseur ou sur le système informatique embarqué est plus sélective que la vérification habituelle de codes pour la machine virtuelle Java telle que décrite dans l'ouvrage intitulé "*The Java Virtual Machine Specification*" précédemment mentionné dans la description.

Toutefois, tout code de la machine virtuelle Java correct au sens du vérificateur Java traditionnel peut

être transformé en un code équivalent susceptible de passer avec succès la vérification de code effectuée sur la carte à microprocesseur.

5 Alors qu'il est possible d'envisager l'écriture directe de codes Java satisfaisant aux critères de vérification précédemment mentionnés dans le cadre de la mise en œuvre du protocole objet de la présente invention, un objet remarquable de celle-ci est également la mise en œuvre d'un procédé de transformation automatique de tout
10 code Java standard en un code normalisé pour le même fragment de programme satisfaisant nécessairement aux critères de vérification mis en œuvre précédemment cités. Le procédé de transformation en code normalisé et le système correspondant seront décrits ultérieurement dans
15 la description de manière détaillée.

Une description plus détaillée du procédé de vérification d'un fragment de programme, ou appliquelette, conforme à l'objet de la présente invention, sera maintenant donnée en liaison avec la figure 3a et les

20 figures suivantes.

D'une manière générale, on indique que le procédé de vérification objet de la présente invention peut être mis en œuvre soit dans le cadre du protocole de gestion d'un fragment de programme objet de l'invention
25 précédemment décrit en liaison avec la figure 2, soit de manière indépendante, afin d'assurer tout processus de vérification jugé nécessaire.

D'une manière générale, on indique qu'un fragment
de programme est constitué par un code objet comportant au
30 moins un sous-programme, plus communément désigné par méthode, et constitué par une suite d'instructions

exécutables par le microprocesseur du système embarqué par l'intermédiaire de la machine virtuelle.

Ainsi que représenté sur la figure 3a, le procédé de vérification consiste, pour chaque sous-programme, à effectuer une étape 200 d'initialisation de la pile des types et du tableau des types de registres de la machine virtuelle par des données représentant l'état de cette machine virtuelle au début de l'exécution du code objet, objet de la vérification. Ce code objet peut être mémorisé temporairement ainsi que décrit précédemment en liaison avec la mise en œuvre du protocole objet de la présente invention.

L'étape 200 précitée est alors suivie d'une étape 200a consistant à positionner la lecture de l'instruction courante I_i , index i , sur la première instruction du code objet. L'étape 200a est suivie d'une étape 201 consistant à effectuer une vérification du code objet précité instruction par instruction, par discrimination pour chaque instruction courante notée I_i de l'existence d'une cible d'instruction de branchement CIB, d'une cible d'un appel de gestionnaire d'exceptions, noté CEM, ou d'une cible d'un appel de sous-routine CSR.

L'étape de vérification 201 est suivie d'une étape de vérification 202 et d'actualisation de l'effet de l'instruction courante I_i sur les types de données de la pile des types et du tableau des types de registres en fonction de l'existence, pour l'instruction courante visée par une autre instruction, d'une cible d'instruction de branchement CIB, d'une cible d'un appel de sous-routine CSR ou d'une cible d'un appel de gestionnaire d'exceptions CEM.

L'étape 202 pour l'instruction courante I_i est suivie d'une étape de test 203 d'atteinte de la dernière instruction, test noté :

I_i = dernière instruction du code objet ?

- 5 Sur réponse négative au test 203, le processus passe à l'instruction suivante 204, noté $i = i+1$, et au retour à l'étape 201.

10 On indique que la vérification précitée, à l'étape 202, est réussie lorsque le tableau des types de registres n'est pas modifié au cours d'une vérification de toutes les instructions I_i constitutives du code objet. Dans ce but, un test 205 d'existence d'un état stable du tableau des types de registres est prévu. Ce test est noté :

3? Etat stable du tableau des types de registres.

- 15 Sur réponse positive au test 205, la vérification a réussi.

Dans le cas où au contraire aucune absence de modification est constatée, le processus de vérification est réitéré et relancé par retour à l'étape 200a. On

- 20 démontre que la fin du processus est garantie après au plus $N_{rx}H$ itérations, où N_r désigne le nombre de registres utilisés et H une constante dépendant de la relation de sous-typage.

- 25 Différentes indications relatives aux types de variables manipulées au cours du processus de vérification décrit précédemment en liaison avec la figure 3a seront maintenant données en liaison avec la figure 3b.

- 30 ~~Les types de variables précitées comportent au~~
moins des identificateurs de classes correspondant aux classes d'objets définis dans le fragment de programme soumis à la vérification, des types de variables

numériques comportant au moins un type short, entier codé sur p bits, p pouvant prendre la valeur $p = 16$, et un type d'adresse de retour d'une instruction de saut JSR, ce type d'adresse étant noté retaddr, un type null relatif à des
 5 références d'objets nuls, un type object relatif aux objets proprement dits, un type spécifique \perp représentant l'intersection de tous les types et correspondant à la valeur zéro nul, un autre type spécifique \top représentant
 10 l'union de tous les types et correspondant à tout type de valeurs.

En référence à la figure 3b, on indique que l'ensemble des types de variables précités vérifient une relation de sous-typage :

object $\in T$;
 15 short,retaddr $\in T$;
 \perp \in null,short,retaddr

Un exemple plus spécifique d'un processus de vérification tel qu'illustré en figure 3a sera maintenant
 20 donné en liaison avec un premier exemple de structure de données illustré au tableau T1 joint en annexe.

L'exemple précité concerne une appliquette écrite en code Java.

Le processus de vérification accède au code du sous-programme constitutif de l'appliquette soumis à
 25 vérification par l'intermédiaire d'un pointeur sur l'instruction I_i en cours de vérification.

Le processus de vérification enregistre la taille et le type de la pile d'exécution au niveau de
 30 l'instruction courante I_i correspondant à saload sur l'exemple du tableau T1 précité.

Le processus de vérification enregistre alors la taille et le type de la pile d'exécution au niveau de l'instruction courante dans la pile des types par l'intermédiaire de son pointeur de pile des types.

5 Ainsi que mentionné précédemment dans la description, cette pile de types reflète l'état de la pile d'exécution de la machine virtuelle au niveau de l'instruction courante I_i . Dans l'exemple représenté au tableau T1, lors de l'exécution future de l'instruction
10 I_i , la pile contiendra trois entrées : une référence vers un objet de la classe C, une référence vers un tableau d'entiers codés sur $p = 16$ bits, le type short[], et un entier de $p = 16$ bits de type short. Ceci est également représenté dans la pile des types qui contient également
15 trois entrées : C, le type des objets de la classe C, short[], le type des tableaux d'entiers $p = 16$ bits et short, le type des entiers $p = 16$ bits.

Une autre structure de données remarquable est constituée par un tableau de types de registres, ce
20 tableau reflétant l'état des registres, c'est-à-dire des registres mémorisant les variables locales, de la machine virtuelle.

En continuant l'exemple indiqué au tableau T1, on indique que l'entrée 0 du tableau de types de registres
25 contient le type C, c'est-à-dire que lors de l'exécution future de l'instruction courante $I_i = \text{saload}$, le registre 0 est assuré de contenir une référence vers un objet de la classe C.

Les différents types manipulés au cours de la
30 vérification et stockés dans le tableau de types de

registres et dans la pile de types sont représentés en figure 3b. Ces types comprennent :

- des identificateurs de classe CB correspondant aux classes d'objets spécifiques définis dans l'appliquette ;
- des types de base, tels que short entier codé sur $p = 16$ bits, int1 et int2, p bits les plus et les moins significatifs respectivement d'entiers codés sur $2p$ bits par exemple, ou retaddr adresse de retour d'une instruction, ainsi que mentionné précédemment ;
- le type null représentant les références d'objets nuls.

En ce qui concerne la relation de sous-typage, on indique qu'un type T1 est sous-type d'un type T2 si toute valeur valide du type T1 est également une valeur valide du type T2. Le sous-typage entre identificateur de classes reflète la hiérarchie d'héritage entre classes de l'appliquette. Sur les autres types, le sous-typage est défini par le treillis représenté en figure 3b, \perp étant sous-type de tous les types et tous les types étant sous-types de \top .

Le déroulement du processus de vérification d'un sous-programme constitutif d'une appliquette est le suivant, en référence au tableau T1 précédemment mentionné.

Le processus de vérification s'effectue indépendamment sur chaque sous-programme de l'appliquette. Pour chaque sous-programme, le processus effectue une ou plusieurs passes de vérification sur les instructions du sous-programme considéré. Le pseudo-code du processus de vérification est donné au tableau T2 joint en annexe.

Le processus de vérification d'un sous-programme débute par l'initialisation de la pile des types et du tableau des types de registres représentés au tableau T1, cette initialisation reflétant l'état de la machine virtuelle au début de l'exécution du sous-programme examiné.

La pile des types est initialement vide, le pointeur de pile est égal à zéro, et les types de registres sont initialisés avec les types des paramètres du sous-programme illustrant le fait que la machine virtuelle passe les paramètres de ce sous-programme dans ces registres. Les types de registres alloués par le sous-programme sont initialisés aux types de données 1 illustrant le fait que la machine virtuelle initialise ces registres à zéro au début de l'exécution du sous-programme.

Ensuite, une ou plusieurs passes de vérification sur les instructions et sur chaque instruction courante I_i du sous-programme sont effectuées.

20 A la fin de la passe de vérification mise en œuvre ou d'une succession de passes par exemple, le processus de vérification détermine si les types de registres contenus dans le tableau des types de registres représentés au tableau T1 de l'annexe ont changé pendant la passe de

25 vérification. En l'absence de changement, la vérification est terminée et un code de succès est renvoyé au programme principal, lequel permet d'envoyer l'accusé de réception positif à l'étape 105 du protocole de gestion représenté

en figure 2.

30 En présence d'une modification du tableau des types de registres précité, le processus de vérification

réitère la passe de vérification jusqu'à ce que les types de registres contenus dans le tableau des types de registres soit stable.

5 Le déroulement proprement dit d'une passe de vérification effectuée une ou plusieurs fois jusqu'à la stabilité du tableau des types de registres sera maintenant décrit en liaison avec les figures 3c à 3j.

Pour chaque instruction courante I_i , les vérifications suivantes sont effectuées :

10 En liaison avec la figure 3a à l'étape 201, le processus de vérification détermine si l'instruction courante I_i est la cible d'une instruction de branchement, d'appel de sous-routine ou d'un gestionnaire d'exception, ainsi que mentionné précédemment. Cette vérification
15 s'effectue par examen des instructions de branchement contenues dans le code du sous-programme et des gestionnaires d'exceptions associés à ce sous-programme.

En référence à la figure 3c ouverte par l'étape 201, lorsque l'instruction courante I_i est la cible d'une
20 instruction de branchement, cette condition étant réalisée par un test 300 désigné par $I_i = \text{CIB}$, ce branchement étant inconditionnel ou conditionnel, le processus de vérification s'assure que la pile des types est vide à ce point du sous-programme par un test 301. Sur réponse
25 positive au test 301, le processus de vérification est poursuivi par une étape suite de contexte noté suite A. Sur réponse négative au test 301, la pile des types n'étant pas vide, la vérification échoue et l'appliquette est rejetée. Cet échec est représentée par l'étape Echec.

30 En référence à la figure 3d ouverte par l'étape suite A, lorsque l'instruction courante I_i est la cible

d'un appel de sous-routine, cette condition étant réalisée par un test 304 $I_1 = \text{CSR}$, le processus de vérification vérifie en un test 305 que l'instruction précédente I_{i-1} ne continue pas en séquence. Cette vérification est réalisée
 5 par une étape de test 305 lorsque l'instruction précédente constitue un branchement inconditionnel, un retour de sous-routine ou une levée d'exception. Le test à l'étape 305 est noté ainsi :

$I_{i-1} = \text{IB}_{\text{inconditionnel}}$, retour RSR ou levée L-EXCEPT.

10 Sur réponse négative au test 305, le processus de vérification échoue en une étape Echec. Au contraire, sur réponse positive au test 305, le processus de vérification réinitialise la pile des types de façon que celle-ci contienne exactement une entrée de type retaddr, adresse
 15 de retour de la sous-routine précitée. Si l'instruction courante I_i à l'étape 304 n'est pas la cible d'un appel de sous-routine, le processus de vérification est poursuivi dans le contexte à l'étape suite B.

En référence à la figure 3e, lorsque l'instruction
 20 courante I_i est la cible d'un gestionnaire d'exceptions, cette condition étant réalisée par un test 307 noté $I_i = \text{CEM}$, CEM désignant la cible d'un gestionnaire d'exceptions, cette condition est réalisée par l'intermédiaire d'un test 307, noté :

25 $I_i = \text{CEM}$.

Le processus, sur réponse positive au test 307 vérifie que l'instruction précédente constitue un branchement
 inconditionnel, un retour de sous-routine ou une levée
 d'exceptions par l'intermédiaire d'un test 305 noté :

30 $I_{i-1} = \text{IB}_{\text{inconditionnel}}$, retour RSR ou levée L-EXCEPT.

Sur réponse positive au test 305, le processus de vérification procède à une réactualisation de la pile des types, à une étape 308, par une entrée de types des exceptions, notée type EXCEPT, l'étape 308 étant suivie d'une étape de suite de contexte, suite C. Sur réponse négative au test 305, le processus de vérification échoue par l'étape notée Echec. Le fragment de programme est alors rejeté.

En référence à la figure 3f, lorsque l'instruction courante I_i est la cible d'une pluralité de branchements incompatibles, cette condition est réalisée par un test 309, lequel est noté :

$$I_i = \text{XIB incompatibles}$$

les branchements incompatibles étant par exemple un branchement inconditionnel et un appel de sous-routine ou encore deux gestionnaires d'exceptions différents. Sur réponse positive au test 309, les branchements étant incompatibles, le processus de vérification échoue par une étape notée Echec et le fragment de programme est rejeté.

Sur réponse négative au test 309, le processus de vérification est poursuivi par une étape notée suite D. Le test 309 est ouvert par l'étape suite C précédemment mentionnée dans la description.

En référence à la figure 3g, lorsque l'instruction courante I_i n'est la cible d'aucun branchement, cette condition étant réalisée par un test 310 ouvert par la suite D précédemment mentionnée, ce test étant noté

$$I_i \exists ? \text{ cibles de branchement,}$$

\exists désignant le symbole d'existence,
le processus de vérification continue sur réponse négative au test 310 par passage à une actualisation de la pile des

types à une étape 311, l'étape 311 et la réponse positive au test 310 étant suivies d'une étape de suite de contexte à l'étape 202, décrite précédemment dans la description en liaison avec la figure 3a.

5 Une description plus détaillée de l'étape de vérification de l'effet de l'instruction courante sur la pile des types à l'étape 202 précédemment citée sera maintenant donnée en liaison avec la figure 3h.

10 Selon la figure précitée, cette étape peut comporter au moins une étape 400 de vérification que la pile d'exécution des types contient au moins autant d'entrées que l'instruction courante comporte d'opérandes. Cette étape de test 400 est notée :

$$Nbep \geq NOpi$$

15 où Nbep désigne le nombre d'entrées de la pile des types et NOpi désigne le nombre d'opérandes contenus dans l'instruction courante.

Sur réponse positive au test 400, ce test est suivi d'une étape 401a de dépilement de la pile des types et de

20 vérification 401b que les types des entrées au sommet de la pile sont sous-types des types des opérandes de l'instruction courante précitée. A l'étape de test 401a, les types des opérandes de l'instruction i sont notés TOpi et les types des entrées au sommet de la pile sont notés

25 Targs.

A l'étape 401b, la vérification correspond à une vérification de la relation de sous-typage Targs sous-type de TOpi.

30 Sur réponse négative au test 400 et au test 401b, le processus de vérification échoue, ce qui est illustré par l'accès à l'étape Echec. Au contraire sur réponse

positive au test 401b, le processus de vérification est poursuivi et consiste à effectuer :

- Une étape de vérification de l'existence d'un espace mémoire suffisant sur la pile des types, pour procéder à l'empilement des résultats de l'instruction courante. Cette étape de vérification est réalisée par un test 402 noté :

$\text{Esp-pile} \geq \text{Esp-résultats}$

où chaque membre de l'inégalité désigne l'espace mémoire correspondant.

Sur réponse négative au test 402, le processus de vérification échoue, ce qui est illustré par l'étape Echec. Au contraire, sur réponse positive au test 402, le processus de vérification procède alors à l'empilement des types de données attribuées aux résultats en une étape 403, l'empilement étant effectué sur la pile des types de données attribuée à ces résultats.

A titre d'exemple non limitatif, on indique que pour la mise en œuvre de la figure 3h de vérification de l'effet de l'instruction courante sur la pile des types, pour une instruction courante constituée par une instruction Java saload correspondant à la lecture d'un élément entier codé sur $p = 16$ bits dans un tableau d'entiers, ce tableau d'entiers étant défini par le tableau d'entiers et un indice entier dans ce tableau, et le résultat par l'entier lu à cet indice dans ce tableau, le processus de vérification s'assure que la pile des types contient au moins deux éléments, que les deux éléments au sommet de la pile des types sont sous-types de short[] respectivement short, procède au processus de

dépilement et ensuite au processus d'empilement du type de données short comme type du résultat.

En outre, en référence à la figure 3i, pour la mise en œuvre de l'étape de vérification de l'effet de l'instruction courante sur la pile des types, lorsque
 5 l'instruction courante I_i est une instruction de lecture, notée IR, d'un registre d'adresse n, cette condition étant réalisée par un test 404 noté $I_i = IR_n$, sur réponse positive au test 404 précité, le processus de vérification consiste
 10 à vérifier le type de données du résultat de cette lecture, en une étape 405, par consultation de l'entrée n du tableau des types de registres, puis à déterminer l'effet de l'instruction courante I_i sur la pile des types par une opération 406a de dépilement des entrées de la
 15 pile correspondant aux opérandes de cette instruction courante et par empilement 406b du type de données de ce résultat. Les opérandes de l'instruction I_i sont notés OP_i . Les étapes 406a et 406b sont suivies d'un retour à la suite de contexte suite F. Sur réponse négative au test
 20 404, le processus de vérification est poursuivi par la suite de contexte suite F.

En référence à la figure 3j, lorsque l'instruction courante I_i est une instruction d'écriture, notée IW, d'un registre d'adresse n, cette condition étant réalisée par
 25 un test noté $I_i = IW_m$, le processus de vérification consiste, sur réponse positive au test 407, à déterminer en une étape 408 l'effet de l'instruction courante sur la pile des types et le type t de l'opérande écrit dans le
 registre d'adresse n, puis, en une étape 409, à remplacer
 30 l'entrée du type du tableau des types de registres à l'adresse n par le type immédiatement supérieur au type

précédemment stocké et au type t de l'opérande écrit dans le registre d'adresse n . L'étape 409 est suivie d'un retour à la suite de contexte suite 204. Sur réponse négative au test 407, le processus de vérification est poursuivi par une suite de contexte suite 204.

A titre d'exemple, lorsque l'instruction courante I_i correspond à l'écriture d'une valeur de type D dans un registre d'adresse 1 et que le type du registre 1 avant vérification de l'instruction était C, le type du registre 1 est remplacé par le type object qui est le type supérieur le plus petit de C et D dans le treillis des types représenté en figure 3b.

De même, à titre d'exemple, lorsque l'instruction courante I_i est une lecture d'une instruction aload-0 consistant à empiler le contenu du registre 0 et que l'entrée 0 du tableau des types de registres est C, le vérificateur empile C sur la pile des types.

Un exemple de vérification d'un sous-programme écrit en environnement Java sera maintenant donné en liaison avec les tableaux T3 et T4 introduits en annexe.

Le tableau T3 représente un code JavaCard spécifique correspondant au sous-programme Java inclus dans ce tableau.

Le tableau T4 illustre le contenu du tableau des types de registres et de la pile des types avant la vérification de chaque instruction. Les contraintes de types sur les opérandes des diverses instructions sont toutes respectées. La pile est vide aussi bien après l'instruction de branchement 5 à l'instruction 9 symbolisée par la flèche, qu'avant la cible de branchement 9 précitée. Le type du registre 1 qui était initialement \perp

devient null, la borne supérieure de null et de 1, lorsque l'instruction 1 de stockage d'une valeur de type null dans le registre 1 est examinée, puis devient de type short[], la borne supérieure du type short[] et du type null,
 5 lorsque l'instruction 8, stockage d'une valeur de type short[] dans le registre 1 est traitée. Le type du registre 1 ayant changé pendant la première passe de vérification, une seconde passe est effectuée en repartant des types de registres obtenus à la fin de la première.
 10 Cette seconde passe de vérification réussit tout comme la première et ne modifie pas les types des registres. Le processus de vérification se termine donc avec succès.

Différents exemples de cas d'échec du processus de vérification sur quatre exemples de code incorrect seront
 15 maintenant donnés en liaison avec le tableau T5 introduit en annexe :

- Au point a) du tableau T5, le code donné en exemple a pour objet de tenter de fabriquer une référence d'objet invalide en utilisant un processus arithmétique sur des
 20 pointeurs. Il est rejeté par la vérification des types des arguments de l'instruction 2 sadd, laquelle exige que ces deux arguments soient de type short.
- Aux points b) et c) du tableau T5, le code a pour objet de réaliser deux tentatives de convertir un entier
 25 quelconque en une référence d'objet. Au point b), le registre 0 est utilisé à la fois avec le type short, instruction 0, et avec le type null, instruction 5. En conséquence, le processus de vérification attribue le
 type I au registre 0 et détecte une erreur de type
 30 lorsque le registre 0 est renvoyé comme résultat de type object à l'instruction 7.

- Au point c) du tableau T5, un ensemble de branchements du type "if...then...else..." est utilisé pour laisser au sommet de la pile un résultat qui est constitué soit par un entier soit par une référence d'objet. Le processus de vérification rejette ce code car il détecte que la pile n'est pas vide au niveau du branchement de l'instruction 5 vers l'instruction 9 symbolisée par la flèche.
- Enfin, au point d) du tableau T5, le code contient une boucle qui, à chaque itération, a pour effet d'empiler un entier de plus au sommet de la pile et de provoquer donc un débordement de la pile après un certain nombre d'itérations. Le processus de vérification rejette ce code en constatant que la pile n'est pas vide au niveau du branchement arrière de l'instruction 8 vers l'instruction 0, symbolisé par la flèche de retour, la pile n'étant pas vide à un point de branchement.

Les différents exemples donnés précédemment en liaison avec les tableaux T3, T4 et T5 montrent que le processus de vérification, objet de la présente invention, est particulièrement efficace et qu'il s'applique à des appliquestes et en particulier aux sous-programmes de ces dernières, pour lesquelles les conditions de type de pile respectivement de caractère vide de la pile des types antérieurement et aux instructions de branchement ou de cibles de branchement sont satisfaites.

Bien entendu, un tel processus de vérification implique l'écriture de codes objet satisfaisant à ces critères, ces codes objet pouvant correspondre au sous-programme introduit au tableau T3 précédemment mentionné.

Toutefois, et afin d'assurer la vérification d'appliquettes et de sous-programmes d'appliquettes existantes qui ne satisfont pas nécessairement aux critères de vérification du procédé objet de la présente invention, en particulier pour ce qui concerne les
 5 appliquettes et les sous-programmes écrits en environnement Java, la présente invention a pour objet d'établir des procédés de transformation de ces
 10 appliquettes ou sous-programmes en appliquettes ou fragments de programme normalisés permettant de subir avec succès les tests de vérification du procédé de vérification objet de la présente invention et du protocole de gestion mettant en œuvre un tel procédé.

Dans ce but, l'invention a donc pour objet la mise
 15 en œuvre d'un procédé et d'un programme de transformation d'un code objet classique constituant une appliquette, ce procédé et ce programme de transformation pouvant être mis en œuvre hors d'un système embarqué ou d'une carte à microprocesseur lors de la création de l'appliquette
 20 considérée.

Le procédé de transformation de code en code normalisé objet de la présente invention, sera maintenant décrit dans le cadre de l'environnement Java à titre d'exemple purement illustratif.

25 Les codes JVM produits par les compilateurs Java existants satisfont à différents critères, lesquels sont énoncés ci-après :

-
- C1 : les arguments de chaque instruction appartiennent
 bien aux types attendus par cette instruction ;
 - 30 C2 : la pile ne déborde pas ;

C'3 : pour chaque instruction de branchement, le type de la pile au niveau de ce branchement est le même qu'au niveau des cibles possibles pour ce branchement ;

- 5 C'4 : une valeur de type t écrite dans un registre en un point du code et relue depuis ce même registre en un autre point du code est toujours relue avec le même type t ;

10 La mise en œuvre du procédé de vérification objet de la présente invention, implique que les critères C'3 et C'4 vérifiés par le code objet soumis à vérification soient remplacés par les critères C3 et C4 ci-après :

C3 : la pile est vide à chaque instruction de branchement et à chaque cible de branchement ;

- 15 C4 : un même registre est utilisé avec un seul et même type dans tout le code d'un sous-programme.

En référence aux critères précités, on indique que les compilateurs Java garantissent seulement les critères ~~plus faibles C'3 et C'4, le processus de vérification~~
20 objet de la présente invention et le protocole de gestion correspondant garantissent en fait des critères C3 et C4 plus contraignants permettant d'assurer la sécurité d'exécution et de gestion des appliquestes.

25 La notion de normalisation recouvrant la transformation des codes en codes normalisés peut présenter différents aspects, dans la mesure où le remplacement des critères C'3 et C'4 par les critères C3 et C4, conformément au processus de vérification objet de la présente invention, peut être réalisé de manière
30 indépendante pour assurer que la pile est vide à chaque instruction de branchement et à chaque cible de

branchement, respectivement que les registres ouverts par l'appliquette sont typés, à chaque registre ouvert correspondant un seul type de donnée attribué pour l'exécution de l'appliquette considérée, ou, au contraire, de manière conjointe, afin de satisfaire à l'ensemble du processus de vérification objet de la présente invention.

Le procédé de transformation d'un code objet en code objet normalisé selon l'invention sera en conséquence décrit selon deux modes de mise en œuvre distincts, un premier mode de mise en œuvre correspondant à la transformation d'un code objet satisfaisant aux critères C1, C2, C'3, C'4 en un code objet normalisé satisfaisant aux critères C1, C2, C3, C'4 correspondant à un code normalisé à instruction de branchement ou cible de branchement vide, puis, selon un deuxième mode de réalisation, dans lequel le code objet classique satisfaisant aux mêmes critères de départ, est transformé en un code objet normalisé satisfaisant aux critères C1, C2, C'3, C4 par exemple correspondant à un code normalisé faisant appel à des registres typés.

Le premier mode de réalisation du procédé de transformation de code, objet de la présente invention, sera maintenant décrit en liaison avec la figure 4a. Dans le mode de réalisation illustré en figure 4a, le code classique de départ est réputé satisfaire aux critères C1+C2+C'3 et le code normalisé obtenu du fait de la transformation est réputé satisfaire aux critères C1+C2+C3.

Selon la figure précitée, le procédé de transformation consiste, pour chaque instruction courante I_i du code ou du sous-programme, à annoter chaque

instruction, en une étape 500, par le type de données de la pile avant et après l'exécution de cette instruction. Les données d'annotation sont notées AI_i et sont associées par la relation $I_i \leftrightarrow AI_i$ en instruction courante considérée.

5 Les données d'annotation sont calculées au moyen d'une analyse du flot des données relatif à cette instruction. Les types de données avant et après exécution de l'instruction sont notés tbe_i et tae_i respectivement. Le calcul des données d'annotation par analyse du flot des
10 données est un calcul classique connu de l'homme du métier et, à ce titre, ne sera pas décrit en détail.

L'opération réalisée à l'étape 500 est illustrée au tableau T6 introduit en annexe dans lequel, pour une appliquette ou sous-programme d'appliquette comportant 12
15 instructions, les données d'annotation AI_i constituées par les types des registres et les types de la pile sont introduites.

L'étape 500 précitée est alors suivie d'une étape 500a consistant à positionner l'index i sur la première
20 instruction $I_1 = I_i$. L'étape 500a est suivie d'une étape 501 consistant à détecter, au sein des instructions et de chaque instruction courante I_i , l'existence de branchements notés IB ou de cibles de branchement CIB pour lesquels la pile d'exécution n'est pas vide. Cette
25 détection 501 est réalisée par un test conduit à partir des données d'annotation AI_i du type des variables de pile alloué à chaque instruction courante, le test étant noté pour l'instruction courante :

I_i est une IB ou CIB et $\text{pile}(AI) \neq \text{vide}$.

30 Sur réponse positive au test 501, c'est-à-dire en présence d'une détection d'une pile d'exécution non vide,

le test précité est suivi d'une étape consistant à insérer des instructions de transfert des variables de pile de part et d'autre de ces branchements IB ou de ces cibles de branchement CIB, afin de vider le contenu de la pile d'exécution dans des registres temporaires avant ce branchement et de rétablir la pile d'exécution à partir des registres temporaires après ce branchement. L'étape d'insertion est notée 502 sur la figure 4a. Elle est suivie d'une étape de test 503 d'atteinte de la dernière instruction noté

I_1 =dernière instruction?

Sur réponse négative au test 503, une incrémentation 504 $i=i+1$ est effectuée pour passage à l'instruction suivante et retour à l'étape 501. Sur réponse positive au test 503, une étape de Fin est lancée. Sur réponse négative au test 501, le procédé de transformation est poursuivi par un branchement vers l'étape 503 en l'absence d'insertion d'instruction de transfert. La mise en œuvre du procédé de transformation d'un code classique en un code normalisé à instruction de branchement à pile vide tel que représenté en figure 4a, permet d'obtenir un code objet normalisé pour le même fragment de programme de départ dans lequel la pile des variables de pile est vide à chaque instruction de branchement et à chaque instruction de cible de branchement, en l'absence de modification de l'exécution du fragment de programme. Dans le cas d'un environnement Java, les instructions de transfert de données entre pile et registre sont les instructions load et store de la machine virtuelle Java.

En reprenant l'exemple introduit au tableau T6, le procédé de transformation détecte une cible de branchement

où la pile n'est pas vide au niveau de l'instruction 9. Il est donc procédé à l'insertion d'une instruction istore 1 avant l'instruction de branchement 5 qui mène à l'instruction 9 précitée afin de sauvegarder le contenu de la pile dans le registre 1 et d'assurer que la pile est vide lors du branchement. Symétriquement, l'insertion d'une instruction iload 1 est effectuée préalablement à la cible d'instruction 9 pour rétablir le contenu de la pile identiquement à ce qu'il était avant le branchement. Finalement, une instruction istore 1 est insérée après l'instruction 8 pour garantir l'équilibre de la pile sur les deux chemins qui mènent à l'instruction 9. Le résultat de la transformation ainsi opérée en un code normalisé est représenté au tableau T7.

Le deuxième mode de réalisation du procédé de transformation objet de la présente invention sera maintenant décrit en liaison avec la figure 4b dans le cas où le code objet classique de départ satisfait aux critères C1+C'4 et le code objet normalisé satisfait aux critères C1+C4.

En référence à la figure 4b précitée, on indique que le procédé, dans ce mode de réalisation, consiste à annoter, selon une étape 500 sensiblement identique à celle représentée en figure 4a, chaque instruction courante I_i par le type de données des registres avant et après l'exécution de cette instruction. De la même manière, les données d'annotations AI_i sont calculées au moyen d'une analyse du flot des données relatif à cette instruction.

L'étape d'annotation 500 est alors suivie d'une étape consistant à effectuer une réallocation des

registres, étape notée 601, par détection des registres d'origine employés avec des types différents, division de ces registres d'origine en registres normalisés distincts, un registre normalisé étant alloué à chaque type de données utilisé. L'étape 601 est suivie d'une étape 602 de réactualisation des instructions qui manipulent les opérandes qui font appel aux registres normalisés précités. L'étape 602 est suivie d'une étape de suite de contexte 302.

10 En référence à l'exemple donné au tableau T6, on indique que le procédé de transformation détecte que le registre de rang 0, noté r0, est utilisé avec les deux types object, instructions 0 et 1, et int, instruction 9 et suivantes. Il est alors procédé à une division du
15 registre d'origine r0 en deux registres, le registre 0 pour l'utilisation des types object et le registre 1 pour les utilisations de type int. Les références au registre 0 de type int sont alors réécrites en les transformant en des références au registre 1, le code normalisé obtenu
20 étant introduit au tableau T8 joint en annexe.

 On note de manière non limitative que dans l'exemple introduit en liaison avec le tableau T8 précité, le nouveau registre 1 est utilisé à la fois pour la normalisation de la pile et pour la création de registres
25 typés par division du registre 0 en deux registres.

 Le procédé de transformation d'un code classique en un code normalisé à instruction de branchement à pile vide tel que décrit en figure 4a sera maintenant décrit de
30 manière plus détaillée dans un mode de réalisation préférentiel non limitatif, en relation avec la figure 5a.

Ce mode de réalisation concerne l'étape 501 consistant à détecter au sein des instructions et de chaque instruction courante I_i l'existence de branchement IB respectivement de cible de branchement CIB pour laquelle la pile n'est pas vide.

Suite à la détermination des instructions cible où la pile n'est pas vide, cette condition étant notée à l'étape 504a, I_i pile≠vide, le processus de transformation consiste à associer, à l'étape 504a précitée, à ces instructions un ensemble de nouveaux registres, un par emplacement de pile actif au niveau de ces instructions. Ainsi, si i désigne le rang d'une cible de branchement dont le type de pile associée n'est pas vide et est du type tpl_i à tpn_i avec $n > 0$, pile non vide, le processus de transformation alloue n nouveau registres, r_1 à r_n non encore utilisés et les associe à l'instruction i correspondante. Cette opération est réalisée à l'étape 504a.

L'étape 504a est suivie d'une étape 504 consistant à examiner chaque instruction détectée de rang i et à discriminer en une étape de test 504 l'existence d'une cible de branchement CIB ou d'un branchement IB. L'étape 504 est représentée sous forme d'un test désigné par :

$\exists ?CIB, IB$ et $I_i = CIB$.

Dans le cas où l'instruction de rang i est une cible de branchement CIB représentée par l'égalité précédente, et que la pile des variables de pile au niveau de cette instruction n'est pas vide, c'est-à-dire en réponse positive au test 504, pour toute instruction précédente de rang $i-1$ constituée par un branchement, une levée d'exceptions ou un retour de programme, cette

condition est réalisée à l'étape de test 505 désignée par :

$I_{i-1} = IB$, levée EXCEPT, retour Prog.

L'instruction détectée de rang i n'est accessible que par un branchement. Sur réponse positive au test 505 précité, le processus de transformation consiste à effectuer une étape 506 consistant à insérer un ensemble d'instructions de chargement du type load à partir de l'ensemble de nouveaux registres antérieurement à l'instruction détectée de rang i considéré. L'opération d'insertion 506 est suivie d'une redirection 507 de tous les branchements vers l'instruction détectée de rang i , vers la première instruction de chargement load insérée. Les opérations d'insertion et de redirection sont représentées au tableau T9 joint en annexe.

Pour toute instruction précédente de rang $i-1$ continuant en séquence, c'est-à-dire lorsque l'instruction courante de rang i est accessible à la fois par un branchement et à partir de l'instruction précédente, cette condition étant réalisée par le test 508 et symbolisée par les relations :

$$I_{i-1} \rightarrow I_i$$

et

$$IB \rightarrow I_i$$

le processus de transformation consiste en une étape 509 à insérer un ensemble d'instructions de sauvegarde store vers l'ensemble de nouveaux registres antérieurement à l'instruction détectée de rang i , et un ensemble d'instructions de chargement load à partir de cet ensemble de nouveaux registres. L'étape 509 est alors suivie d'une étape 510 de redirection de tous les branchements vers

l'instruction détectée de rang i vers la première instruction de chargement load insérée.

Dans le cas où l'instruction détectée de rang i est un branchement vers une instruction déterminée, pour
 5 toute instruction détectée de rang i constituée par un branchement inconditionnel, cette condition étant réalisée par un test 511 noté :

$$I_i = IB_{\text{incondit.}}$$

le processus de transformation tel que représenté en
 10 figure 5a consiste à insérer à une étape 512, sur réponse positive au test 511, antérieurement à l'instruction détectée de rang i , une pluralité d'instructions de sauvegarde store. Le processus de transformation insère
 15 avant l'instruction i les n instructions store ainsi que représenté en exemple au tableau T11. Les instructions store adressent les registres r_1 à r_n , n désignant le nombre de registres. A chaque nouveau registre est ainsi associée l'instruction de sauvegarde.

Pour toute instruction détectée de rang i
 20 constituée par un branchement conditionnel et pour un nombre mOp supérieur à 0 d'opérandes manipulés par cette instruction de branchement conditionnel, cette condition étant réalisée par le test 513 noté :

$$I_i = IB_{\text{condit.}}$$

25 avec $mOp > 0$

le processus de transformation, en réponse positive au test 513 précité, consiste à insérer à une étape 514
~~antérieurement à cette instruction détectée de rang i , une~~
 instruction de permutation notée swap_x au sommet de la
 30 pile des variables de pile des mOp opérandes de l'instruction détectée de rang i et des n valeurs

suivantes. Cette opération de permutation permet de ramener au sommet de la pile des variables de pile les n valeurs à sauvegarder dans l'ensemble des nouveaux registres r_1 à r_n . L'étape 514 est suivie d'une étape 515 consistant à insérer antérieurement à l'instruction de rang i un ensemble d'instructions de sauvegarde store vers l'ensemble des nouveaux registres r_1 à r_n . L'étape 515 d'insertion précitée est elle-même suivie d'une étape 516 d'insertion postérieurement à l'instruction détectée de rang i d'un ensemble d'instructions de chargement load à partir de l'ensemble des nouveaux registres r_1 à r_n . L'ensemble des opérations d'insertion correspondantes est représenté au tableau 12 introduit en annexe.

Pour des raisons de complétude et en référence à la figure 5a, on indique que, sur réponse négative au test 504, la poursuite du processus de transformation est réalisée par une étape de suite de contexte suite 503, que la réponse négative aux tests, 505, 508, 511 et 513 est elle-même suivie d'une poursuite du processus de transformation par l'intermédiaire d'une étape de suite de contexte suite 503 et qu'il en est de même en ce qui concerne la poursuite des opérations après les étapes de redirection 507 et 510 et d'insertion 512 et 516 précitées.

Une description plus détaillée du procédé de normalisation et de transformation d'un code objet en un code objet normalisé faisant appel à des registres typés tel que décrit en figure 4b, sera maintenant donnée en liaison avec la figure 5b. Ce mode de réalisation concerne plus particulièrement un mode de réalisation préférentiel non limitatif de l'étape 601 de réallocation des registres

par détection des registres d'origine employés avec des types différents.

En référence à la figure 5b précitée, on indique que l'étape 601 précitée consiste à déterminer en une
 5 étape 603 les intervalles de durée de vie notés ID_j de chaque registre r_j . Ces intervalles de durée de vie, désignés par "live range" ou "webs" en langage anglo-saxon, sont définis pour un registre r comme un ensemble maximal de traces partielles tel que le registre r est
 10 vivant en tous points de ces traces. Pour une définition plus détaillée de ces notions, on pourra utilement se reporter à l'ouvrage édité par Steven S. MUCHNICK intitulé "Advanced Compiler Design and Implementation", Section 16.3, Morgan KAUFMANN, 1997. L'étape 603 est désignée par
 15 la relation :

$$ID_j \leftrightarrow r_j$$

selon laquelle à chaque registre r_j est associé un intervalle de durée de vie ID_j correspondant.

L'étape 603 précitée est suivie d'une étape 604
 20 consistant à déterminer, à l'étape 604, le type de données principal, noté tp_j , de chaque intervalle de durée de vie ID_j . Le type principal d'un intervalle de durée de vie ID_j , pour un registre r_j , est défini par la borne supérieure des types de donnée stockés dans ce registre r_j ,
 25 par les instruction de sauvegarde store appartenant à l'intervalle de durée de vie précité.

L'étape 604 est elle-même suivie d'une étape 605
 consistant à établir un graphe d'interférences entre les
 intervalles de durée de vie précédemment définis aux
 30 étapes 603 et 604, ce graphe d'interférences consistant en un graphe non orienté dont chaque sommet est constitué par

un intervalle de durée de vie et dont les arcs, notés a_{j_1, j_2} sur la figure 5b, entre deux sommets ID_{j_1} et ID_{j_2} , existent si un sommet contient une instruction de sauvegarde adressée au registre de l'autre sommet ou réciproquement. Sur la figure 5b, la construction du graphe d'interférences est représentée symboliquement, cette construction pouvant être réalisée à partir de techniques de calcul connues de l'homme du métier. Pour une description plus détaillée de la construction de ce type de graphe, on pourra utilement se reporter à l'ouvrage publié par Alfred V. AHO, Ravi SETHI et Jeffrey D. ULLMAN intitulé *"Compilers : principes, techniques, and tools"*, Addison-Wesley 1986, section 9.7.

Suite à l'étape 605, le procédé de normalisation tel que représenté en figure 5b consiste à traduire à une étape 606 l'unicité d'un type de donnée alloué à chaque registre r_j dans le graphe d'interférences en ajoutant des arcs entre toutes paires de sommets du graphe d'interférences tant que deux sommets d'une paire de sommets n'ont pas le même type de données principal associé. On comprend que la traduction du caractère d'unicité d'un type de donnée alloué à chaque registre correspond bien entendu à la traduction et à la prise en compte du critère C4 dans le graphe d'interférences, critère mentionné précédemment dans la description. L'étape 606 précitée est alors suivie d'une étape 607 dans laquelle une instanciation du graphe d'interférences est effectuée, instanciation plus communément désignée par étape de coloriage du graphe d'interférences selon les techniques habituelles. Au cours de l'étape 607, le processus de transformation attribue à chaque intervalle

de durée de vie ID_{jk} un numéro de registre rk de telle manière que deux intervalles adjacents dans le graphe d'interférences reçoivent des numéros de registres différents.

5 Cette opération peut être réalisée à partir de tout processus adapté. A titre d'exemple non limitatif, on indique qu'un processus préférentiel peut consister :

10 a) à choisir un sommet de degré minimal dans le graphe d'interférences, le degré minimal étant défini comme un nombre de sommets adjacents minimal, et de le retirer du graphe. Cette étape peut être répétée jusqu'à ce que le graphe soit vide.

15 b) Chaque sommet précédemment retiré est réintroduit dans le graphe d'interférences dans l'ordre inverse de leur retrait, le dernier enlevé étant le premier réintroduit et successivement dans l'ordre inverse de l'ordre de retrait. Ainsi, à chaque sommet réintroduit, peut être attribué le plus petit numéro de registre qui est différent des numéros attribués à

20 tous les sommets adjacents.

Enfin, par l'étape 602, représentée en figure 4b, le processus de transformation et de réallocation réécrit les instructions d'accès aux registres figurant dans le code du sous-programme de l'appliquette considérée. Un accès à

25 un registre donné dans l'intervalle de durée de vie correspondant est remplacé par un accès à un registre différent dont le numéro a été attribué pendant la phase d'instanciation encore désignée par phase de coloriage.

30 Une description plus détaillée d'un système informatique embarqué permettant la mise en œuvre du protocole de gestion et du processus de vérification d'un

fragment de programme ou appliquette conforme à l'objet de la présente invention et d'un système de développement d'une appliquette sera maintenant donnée en liaison avec la figure 6.

5 En ce qui concerne le système embarqué correspondant portant la référence 10, on rappelle que ce système embarqué est un système du type reprogrammable comportant les éléments essentiels tels que représentés en figure 1b. Le système embarqué précité est réputé
10 interconnecté à un terminal par une liaison série, le terminal étant lui-même relié par exemple par l'intermédiaire d'un réseau local, le cas échéant d'un réseau lointain, à un ordinateur de développement de l'appliquette portant la référence 20. Sur le système
15 embarqué 10 fonctionne un programme principal qui lit et exécute les commandes envoyées sur la liaison série par le terminal. En outre, les commandes standard pour une carte à microprocesseur, telles que par exemple les commandes standard du protocole ISO 7816, peuvent être mises en
20 œuvre, le programme principal reconnaissant de plus deux commandes supplémentaires, l'une pour le téléchargement d'une appliquette, et l'autre pour la sélection d'une appliquette préalablement chargée sur la carte à microprocesseur.

25 Conformément à l'objet de la présente invention, la structure du programme principal est réalisée de manière à comporter au moins un module de programme de ~~gestion et de vérification d'un fragment de programme~~
téléchargé suivant le protocole de gestion d'un fragment
30 de programme téléchargé précédemment décrit dans la description en liaison avec la figure 2.

En outre, le module de programme comporte également un module de sous-programme de vérification d'un fragment de programme téléchargé suivant le procédé de vérification tel que décrit précédemment dans la description en liaison avec les figures 3a à 3j.

Dans ce but, la structure des mémoires, en particulier de la mémoire permanente non inscriptible, mémoire ROM, est modifiée de manière à comporter notamment, outre le programme principal, un module 17 de gestion de protocole et de vérification, ainsi que mentionné précédemment. Enfin, en ce qui concerne la mémoire non volatile réinscriptible de type EEPROM, celle-ci comporte avantageusement un répertoire d'appliquettes, noté 18, permettant la mise en œuvre du protocole de gestion et du processus de vérification objets de la présente invention.

En référence à la même figure 6, on indique que le système de développement de l'appliquette conforme à l'objet de la présente invention, permettant en fait la transformation d'un code objet classique ainsi que mentionné précédemment dans la description et satisfaisant aux critères $C1+C2+C'3+C'4$ dans le cadre de l'environnement Java en un code objet normalisé pour le même fragment de programme comprend, associé à un compilateur classique Java 21, un module de transformation de code, noté 22, lequel procède à la transformation de code en code normalisé selon le premier et le deuxième mode de réalisation précédemment décrits dans la description en liaison avec les figures 4a, 4b et 5a, 5b. On comprend en effet que, d'une part, la normalisation du code objet d'origine en un code objet normalisé à

instruction de branchement à pile vide et en un code normalisé faisant appel à des registres typés, d'autre part, ainsi que mentionné précédemment dans la description, permet de satisfaire aux critères de
5 vérification C3 et C4 imposés par le procédé de vérification objet de la présente invention.

Le module de transformation de code 22 est suivi d'un convertisseur JavaCard 23, lequel permet d'assurer la transmission par un réseau distant ou local vers le
10 terminal et, par l'intermédiaire de la liaison série, vers la carte à microprocesseur 10. Ainsi, le système de développement de l'appliquette 20 représenté en figure 6 permet de transformer les fichiers de classe compilés produits par le compilateur Java 21 à partir des codes
15 source Java de l'appliquette en des fichiers de classe équivalents mais qui respectent les contraintes supplémentaires C3, C4 imposées par le protocole de gestion et le module de vérification 17 embarqués sur la carte à microprocesseur 10. Ces fichiers de classe
20 transformés sont convertis en une appliquette téléchargeable sur la carte par le convertisseur JavaCard standard 23.

Différents éléments particulièrement remarquables de l'ensemble des éléments du protocole, des procédés et
25 des systèmes objets de la présente invention, seront maintenant donnés à titre indicatif.

Vis-à-vis des processus de vérification de l'art antérieur tels que mentionnés dans l'introduction à la description, le procédé de vérification objet de la
30 présente invention apparaît remarquable en ce qu'il concentre l'effort de vérification sur les propriétés de

typage des opérandes qui sont essentielles à la sécurité
 de l'exécution de chaque appliquette, c'est-à-dire du
 respect des contraintes de type associées à chaque
 instruction et absence de débordement de pile. D'autres
 5 vérifications n'apparaissent pas essentielles en termes de
 sécurité, en particulier la vérification que le code
 initialise correctement chaque registre avant de le lire
 pour la première fois. Le procédé de vérification objet de
 la présente invention opère au contraire par
 10 initialisation à zéro de tous les registres à partir de la
 machine virtuelle lors de l'initialisation de la méthode
 afin de garantir que la lecture d'un registre non
 initialisé ne peut compromettre la sécurité de la carte.

En outre, l'exigence imposée par le procédé de
 15 vérification objet de la présente invention selon laquelle
 la pile doit être vide à chaque instruction de branchement
 ou de cible de branchement, garantit que la pile est dans
 le même état, vide, après exécution du branchement et
 avant exécution de l'instruction à laquelle le programme
 20 s'est branché. Ce mode opératoire garantit que la pile est
 dans un état cohérent, quel que soit le chemin d'exécution
 suivi à travers le code du sous-programme ou de
 l'appliquette considéré. La cohérence de la pile est ainsi
 garantie, même en présence de branchement ou de cible de
 25 branchement. Contrairement aux procédés et aux systèmes de
 l'art antérieur, dans lesquels il est nécessaire de
 conserver en mémoire vive le type de la pile à chaque
 30 ~~cible de branchement, ce qui nécessite une quantité de~~
 mémoire vive proportionnelle à $TpxNb$, produit de la taille
 maximale de pile d'exécution utilisée et du nombre de
 cibles de branchement dans le code, le procédé de

vérification, objet de la présente invention, n'a besoin que du type de la pile d'exécution lors de l'instruction en cours de vérification et il ne conserve pas en mémoire le type de cette pile à d'autres points du code. En
 5 conséquence, le procédé objet de l'invention se contente d'une quantité de mémoire vive proportionnelle à T_p mais indépendante de N_b , et par conséquent de la longueur du code du sous-programme ou de l'appliquette.

L'exigence selon le critère C4, selon lequel un
 10 registre donné doit être utilisé avec un seul et même type dans tout le code d'un sous-programme, garantit que le code précité n'utilise pas un registre de manière incohérente, par exemple en y écrivant un entier short à un point du programme et en le relisant comme une
 15 référence d'objet à un autre point du programme.

Dans les processus de vérification décrits dans l'art antérieur, en particulier dans la spécification Java intitulée "*The Java Virtual Machine Specification*" éditée par Tim LINDHOLM et Frank YELLIN, déjà citée, pour
 20 ~~garantir la cohérence des utilisations précitées à travers~~ les instructions de branchement, il est nécessaire de conserver en mémoire vive une copie du tableau des types de registres à chaque cible de branchement. Cette opération nécessite une quantité de mémoire vive
 25 proportionnelle à $T_r \times N_b$ où T_r désigne le nombre de registres utilisés par le sous-programme et N_b le nombre de cibles de branchement dans le code de ce sous-programme.

Au contraire, le processus de vérification objet
 30 de la présente invention, opère sur un tableau global de types de registres en l'absence de conservation en mémoire

vive de copie à différents points du code. En conséquence, la mémoire vive nécessaire pour réaliser le processus de vérification est proportionnelle à T_r mais indépendante de N_b et donc de la longueur du code du sous-programme considéré.

La contrainte selon laquelle un registre donné est utilisé avec le même type en tous points, c'est-à-dire en toute instruction du code considéré, simplifie sensiblement et de manière significative la vérification des sous-programmes. Au contraire, dans les processus de vérification de l'art antérieur, en l'absence d'une telle contrainte, le processus de vérification doit établir que les sous-programmes respectent une discipline de pile stricte et doit vérifier le corps des sous-programmes de manière polymorphe en ce qui concerne le type de certains registres.

En conclusion, le processus de vérification objet de la présente invention vis-à-vis des techniques de l'art antérieur permet, d'une part, de réduire la taille du code du programme permettant de conduire le procédé de vérification, et, d'autre part, de réduire la consommation de mémoire vive lors des opérations de vérification, le degré de complexité étant de la forme $O(T_p + P_r)$ dans le cas du processus de vérification objet de la présente invention, au lieu de $(O(T_p + T_r) \times N_b)$ pour ce qui concerne les processus de vérification de l'art antérieur, en offrant toutefois les mêmes garanties vis-à-vis de la sûreté de l'exécution du code vérifié.

Enfin, le processus de transformation d'un code classique d'origine en un code normalisé est réalisé par transformation localisée du code en l'absence de

transmission d'informations supplémentaires à l'organe vérificateur, c'est-à-dire à la carte à microprocesseur ou au système informatique embarqué.

En ce qui concerne le procédé de réallocation des registres tel que décrit en figures 4b et 5b, ce procédé se distingue des procédés connus de l'art antérieur décrits notamment par le brevet US 4,571,678 et par le brevet US 5,249,295, par le fait :

- que la réallocation de registre assure qu'un même registre ne peut être attribué à deux intervalles possédant des types principaux différents, ce qui garantit ainsi qu'un registre donné est utilisé avec le même type dans tout le code ; et

- que les algorithmes d'allocations de registres existants et décrits dans les documents précités supposent un nombre fixe de registres et tentent de minimiser les transferts désignés par "*spills*" en langage anglo-saxon, entre registres et pile, alors que la réallocation des registres conformément à l'objet de

la présente invention opère dans un cadre où le nombre total de registres est variable, en conséquence de quoi il n'y a pas lieu d'effectuer des transferts entre registres et piles alors qu'un processus de minimisation du nombre total de registres est mis en œuvre.

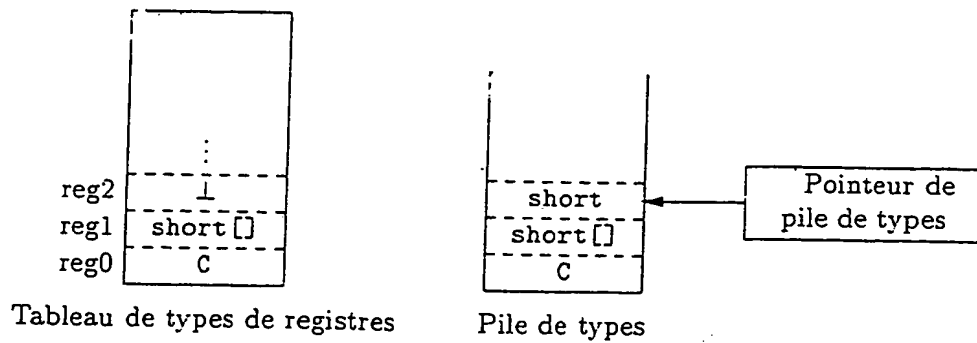
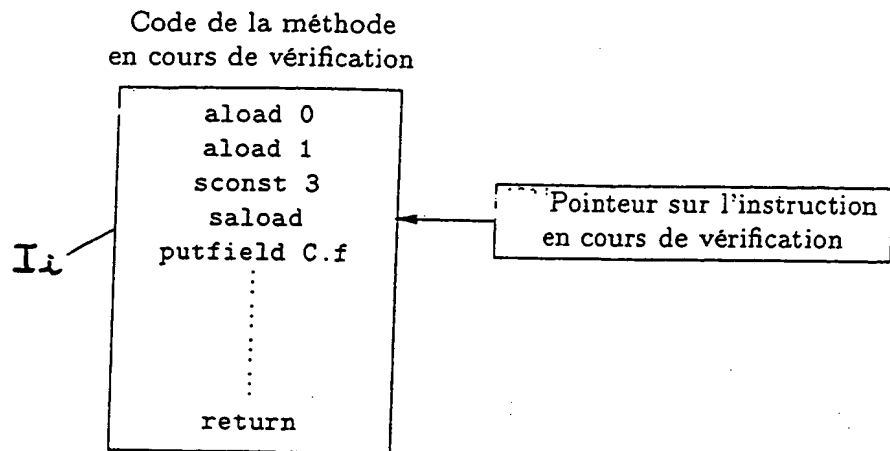
ANNEXESTABEAU T1

TABLEAU T2PSEUDO-CODE DU MODULE VERIFICATEUR

Variables globales utilisées:

T_r	nombre de registres déclaré par la méthode courante
T_p	taille maximale de pile déclarée par la méthode courante
$tr[T_r]$	tableau de types de registres (402 dans la figure 4)
$tp[T_p]$	type de pile (403 dans la figure 4)
pp	pointeur de pile (404 dans la figure 4)
chg	drapeau indiquant si tr a changé

Initialiser $pp \leftarrow 0$ Initialiser $tp[0] \dots tp[n-1]$ à partir des types des n arguments de la méthodeInitialiser $tp[n] \dots tp[T_r-1]$ à \perp Initialiser chg à vraiTant que chg est vrai:Remettre chg à faux

Se positionner sur la première instruction de la méthode

Tant que la fin de la méthode n'est pas atteinte:

Si l'instruction courante est la cible d'une instruction de branchement:

Si $pp \neq 0$, échec de la vérification

Si l'instruction courante est la cible d'un appel de sous-routine:

Si l'instruction précédente continue en séquence, échec

Prendre $tp[0] \leftarrow \text{retaddr}$ et $pp \leftarrow 1$ Si l'instruction courante est un gestionnaire d'exceptions de classe C :

Si l'instruction précédente continue en séquence, échec

Faire $tp[0] \leftarrow C$ et $pp \leftarrow 1$

Si l'instruction courante est une cible de sortes différentes:

Échec de la vérification

Déterminer les types a_1, \dots, a_n des arguments de l'instructionSi $pp < n$, échec (débordement de pile)Pour $i = 1, \dots, n$:Si $tp[pp-n-i-1]$ n'est pas sous-type de a_i , échecFaire $pp \leftarrow pp - n$ Déterminer les types r_1, \dots, r_m des résultats de l'instructionSi $pp + m \geq T_p$, échec (débordement de pile)Pour $i = 1, \dots, m$, faire $tp[pp+i-1] \leftarrow r_i$ Faire $pp \leftarrow pp + m$ Si l'instruction courante est une écriture dans un registre r :Déterminer le type t de la valeur écrite dans le registreFaire $tr[r] \leftarrow \text{borne inférieure}(t, tr[r])$ Si $tr[r]$ a changé, faire $chg \leftarrow \text{vrai}$

Si l'instruction courante est un branchement:Si $pp \neq 0$, échec de la vérification

Avancer à la prochaine instruction

Renvoyer un code de succès de la vérification

TABLEAU T3

```

static short[] meth(short [] tableau)
{
    short[] resultat = null;
    if (tableau.length >= 2) resultat = tableau;
    return tableau;
}

```

TABLEAU T4

Première itération sur le code de la méthode:

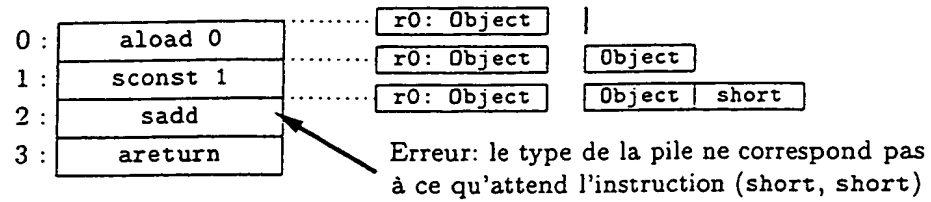
Code de la méthode	Tableau des types des registres	Type de la pile
0: aconst_null	r0: short[] r1: <u>1</u>	
1: astore 1	r0: short[] r1: <u>1</u>	null
2: aload 0	r0: short[] r1: null	
3: arraylength	r0: short[] r1: null	short[]
4: sconst 2	r0: short[] r1: null	short
5: if_scmlt 9	r0: short[] r1: null	short short
7: aload 0	r0: short[] r1: null	
8: astore 1	r0: short[] r1: null	short[]
9: aload 1	r0: short[] r1: short[]	
10: areturn	r0: short[] r1: short[]	short[]

Seconde itération sur le code de la méthode:

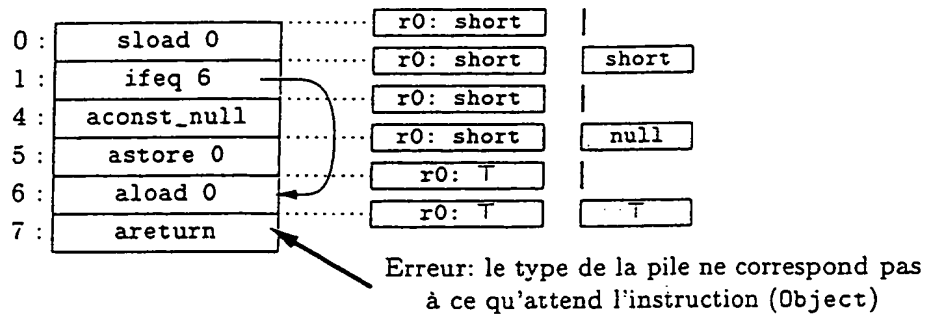
0: aconst_null	r0: short[] r1: short[]	
1: astore 1	r0: short[] r1: short[]	null
2: aload 0	r0: short[] r1: short[]	
3: arraylength	r0: short[] r1: short[]	short[]
4: sconst 2	r0: short[] r1: short[]	short
5: if_scmlt 9	r0: short[] r1: short[]	short short
7: aload 0	r0: short[] r1: short[]	
8: astore 1	r0: short[] r1: short[]	short[]
9: aload 1	r0: short[] r1: short[]	
10: areturn	r0: short[] r1: short[]	short[]

TABLEAU T5

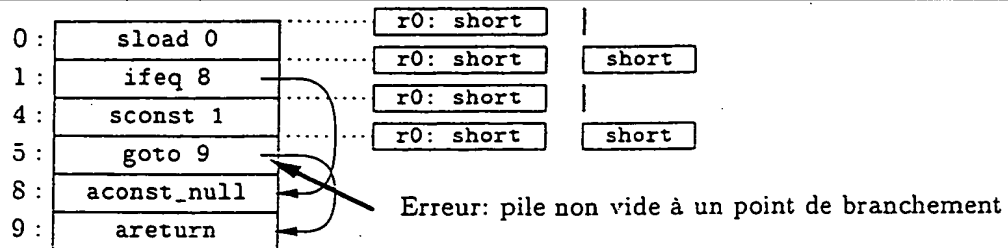
(a) *Violation des contraintes de types sur les arguments d'une instruction:*



(b) *Utilisation incohérente d'un registre:*



(c) *Branchements introduisant des incohérences au niveau de la pile:*



(d) *Débordement de pile à l'intérieur d'une boucle:*

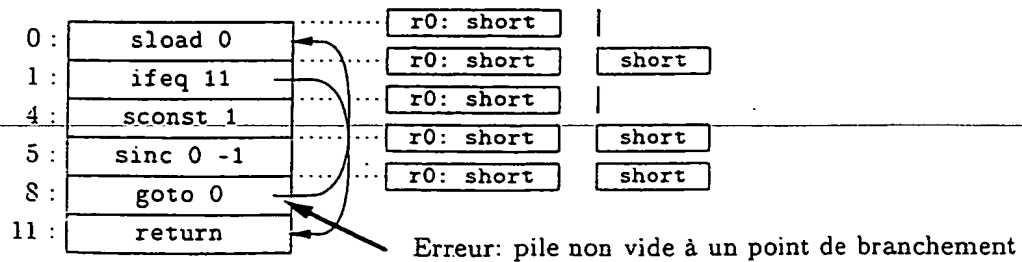


TABLEAU T6

(a) Code initial de la méthode, annoté par les types des registres et de la pile:

0:	aload 0	r0: Object	
1:	ifnull 8	r0: Object	Object
4:	iconst 1	r0: Object	
5:	goto 9	r0: Object	int
8:	iconst 0	r0: Object	
9:	ineg	r0: Object	int
10:	istore 0	r0: Object	int
11:	iload 0	r0: int	
12:	ireturn	r0: int	int

TABLEAU T7

(b) Code de la méthode après normalisation de la pile au niveau du branchement 5 → 9:

0:	aload 0	r0: Object	r1: ⊥	
1:	ifnull 8	r0: Object	r1: ⊥	Object
4:	iconst 1	r0: Object	r1: ⊥	
4':	istore 1	r0: Object	r1: ⊥	int
5:	goto 8''	r0: Object	r1: int	
8:	iconst 0	r0: Object	r1: int	
8':	istore 1	r0: Object	r1: ⊥	int
8'':	iload 1	r0: Object	r1: int	
9:	ineg	r0: Object	r1: int	int
10:	istore 0	r0: Object	r1: int	int
11:	iload 0	r0: int	r1: int	
12:	ireturn	r0: int	r1: int	int

TABLEAU T8

(c) Code de la méthode après réallocation des registres:

0 :	aload 0	r0: Object	r1: \perp	
1 :	ifnull 8	r0: Object	r1: \perp	Object
4 :	iconst 1	r0: Object	r1: \perp	
4' :	istore 1	r0: Object	r1: int	int
5 :	goto 8''	r0: Object	r1: int	
8 :	iconst 0	r0: Object	r1: \perp	int
8' :	istore 1	r0: Object	r1: int	
8'' :	iload 1	r0: Object	r1: int	int
9 :	ineg	r0: Object	r1: int	int
10 :	istore 1	r0: Object	r1: int	
11 :	iload 1	r0: Object	r1: int	int
12 :	ireturn			

TABLEAU T9

(a) Cible de branchement, instruction précédente ne continuant pas en séquence:

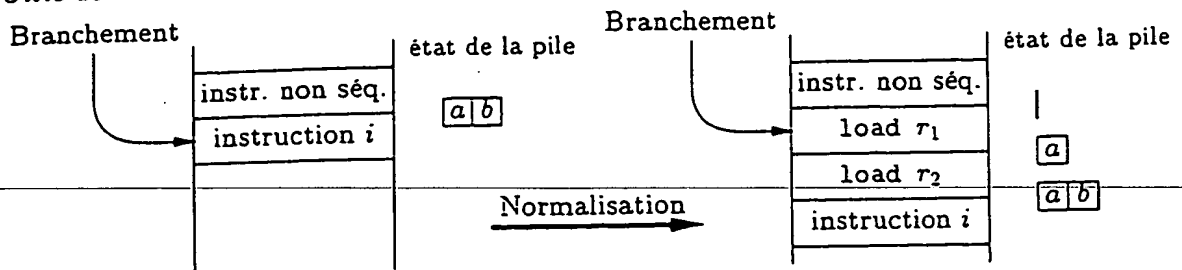
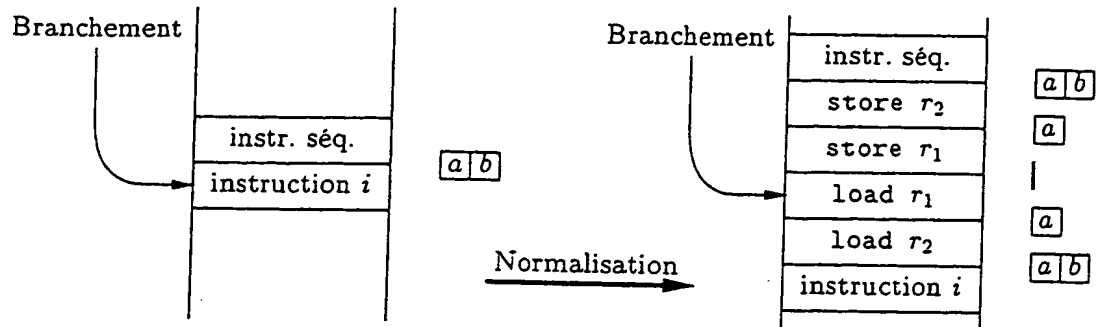
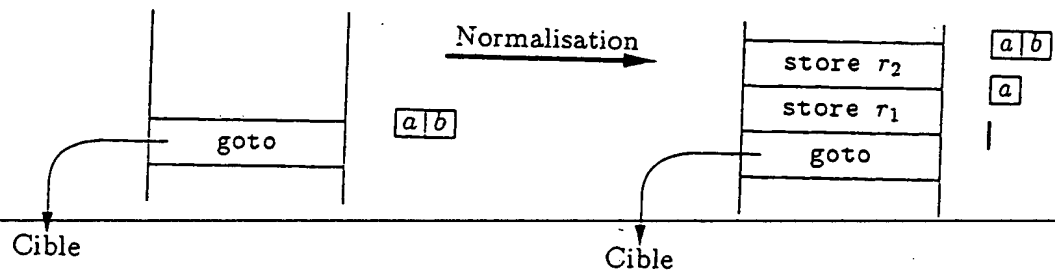


TABLEAU T10

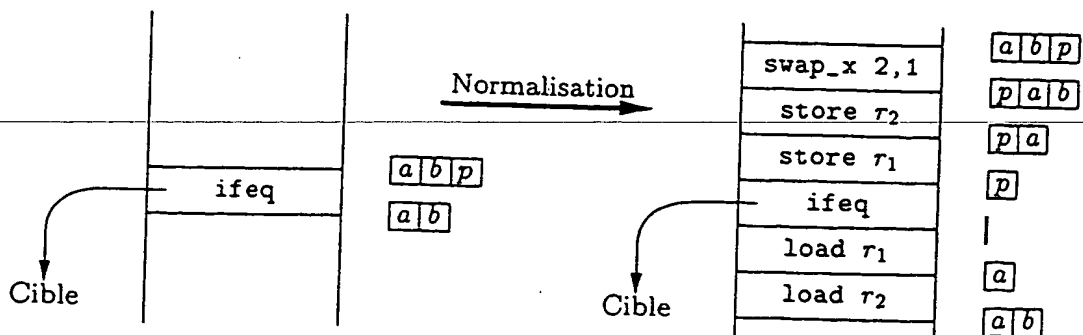
(b) *Cible de branchement, instruction précédente continuant en séquence:*

**TABLEAU T11**

(c) *Branchement inconditionnel sans arguments:*

**TABLEAU T12**

(d) *Branchement conditionnel à un argument:*



REVENDICATIONS

1. Protocole de gestion d'un fragment de programme téléchargé sur un système embarqué reprogrammable, tel qu'une carte à microprocesseur munie d'une mémoire réinscriptible, ledit fragment de programme étant
5 constitué par un code objet, suite d'instructions, exécutable par le microprocesseur du système embarqué par l'intermédiaire d'une machine virtuelle munie d'une pile d'exécution et de registres ou variables locales
10 manipulées par ces instructions et permettant d'interpréter ce code objet, ledit système embarqué étant interconnecté à un terminal, caractérisé en ce que ce protocole consiste au moins, au niveau dudit système embarqué :
- 15 a) à détecter une commande de téléchargement de ce fragment de programme ; et sur réponse positive à cette étape consistant à détecter une commande de téléchargement,
- b) à lire le code objet constitutif de ce fragment de
20 ~~programme et à mémoriser temporairement ce code objet~~
dans ladite mémoire réinscriptible ;
- c) à soumettre l'ensemble du code objet mémorisé temporairement à un processus de vérification instruction par instruction, ce processus de
25 vérification consistant au moins en une étape d'initialisation de la pile des types et du tableau des types de registres, représentant l'état de ladite machine virtuelle au début de l'exécution du code
objet mémorisé temporairement et en une succession
30 d'étapes de vérification instruction par instruction, par discrimination de l'existence, pour chaque

instruction courante, d'une cible, cible d'instruction de branchement, cible d'un appel d'un gestionnaire d'exceptions ou cible d'un appel de sous-routine et par une vérification et une actualisation de l'effet de ladite instruction courante sur la pile des types et sur le tableau des types de registres, et, dans le cas d'une vérification réussie dudit code objet,

- d) à enregistrer le fragment de programme téléchargé dans un répertoire de fragments de programmes disponibles, et
- e) à renvoyer audit lecteur un accusé de réception positif ; et, dans le cas d'une vérification non réussie dudit code objet,
- f) à effacer le fragment de programme enregistré momentanément, en l'absence d'enregistrement de ce dernier dans ledit répertoire de fragments de programmes disponibles, et
- g) à adresser audit lecteur un code d'erreur.

2. Protocole selon la revendication 1, caractérisé

-
- 20 en ce que, sur réponse négative à ladite étape a) consistant à détecter une commande de téléchargement, celui-ci consiste :
 - b') à détecter une commande de sélection d'un fragment de programme disponible dans un répertoire de fragments de programmes ; et, sur réponse positive à cette étape consistant à détecter une commande de sélection d'un fragment de programme disponible ;
 - c') à appeler ledit fragment de programme disponible sélectionné ;
 - 30 d') à exécuter ledit fragment de programme disponible appelé par l'intermédiaire de la machine virtuelle, en
-

l'absence de toute vérification dynamique de types de variables, des droits d'accès aux objets manipulés par le fragment de programme disponible appelé, du débordement de la pile d'exécution lors de l'exécution de chaque instruction, et, sur réponse négative à cette étape consistant à détecter une commande de sélection d'un fragment de programme disponible, e') à procéder au traitement des commandes standards du système embarqué.

3. Procédé de vérification d'un fragment de programme téléchargé sur un système embarqué reprogrammable, tel qu'une carte à microprocesseur munie d'une mémoire réinscriptible, ledit fragment de programme étant constitué par un code objet et comportant au moins un sous-programme, suite d'instructions, exécutable par le microprocesseur du système embarqué par l'intermédiaire d'une machine virtuelle munie d'une pile d'exécution et de registres d'opérandes manipulés par ces instructions et permettant d'interpréter ce code objet, ledit système

embarqué étant interconnecté à un lecteur, caractérisé en ce que ledit procédé consiste, suite à la détection d'une commande de téléchargement et à la mémorisation dudit code objet constitutif de ce fragment de programme dans ladite mémoire réinscriptible, pour chaque sous-programme :

α) à effectuer une étape d'initialisation de la pile des types et du tableau des types de registres par des données représentant l'état de la machine virtuelle au début de l'exécution du code objet mémorisé temporairement ;

β) à effectuer une vérification dudit code objet mémorisé temporairement instruction par instruction, par

discrimination de l'existence, pour chaque instruction courante, d'une cible, cible d'instruction de branchement, cible d'un appel d'un gestionnaire d'exceptions ou cible d'un appel de sous-routine ;

- 5 γ) à effectuer une vérification et une actualisation de l'effet de ladite instruction courante sur les types de données de ladite pile des types et dudit tableau des types de registres, en fonction de l'existence d'une cible d'instruction de branchement, d'une cible
10 d'un appel de sous-routine ou d'une cible d'un appel de gestionnaire d'exceptions, ladite vérification étant réussie lorsque le tableau des types de registres n'est pas modifié au cours d'une
15 vérification de toutes les instructions et le processus de vérification étant poursuivi instruction par instruction jusqu'à ce que le tableau des types de registres soit stable, en l'absence de modification, le processus de vérification étant interrompu sinon.

4. Procédé de vérification selon la revendication

- 20 3, caractérisé en ce que les types de variables manipulées au cours du processus de vérification comprennent au moins :
- des identificateurs de classes correspondant aux classes d'objets définies dans le fragment de
25 programme ;
 - des types de variables numériques comportant au moins un type short, entier codé sur p bits, et un type retaddr d'adresse de retour d'une instruction de saut JSR ;
 - 30 - un type null relatif à des références d'objets nulles ;
 - un type object relatif aux objets ;

- un premier type spécifique 1, représentant l'intersection de tous les types et correspondant à la valeur 0, nil ;
- un deuxième type spécifique T, représentant l'union de tous les types et correspondant à tout type de valeur.

5 5. Procédé selon la revendication 4, caractérisé en ce que l'ensemble desdits types de variables vérifie une relation de sous-typage :

object \in T;

10 short, retaddr \in T;

1 \in null, short, retaddr.

15 6. Procédé selon l'une des revendications 3 à 5, caractérisé en ce que lorsque ladite instruction courante est la cible d'une instruction de branchement, ledit procédé de vérification consiste à vérifier que la pile des types est vide, le processus de vérification étant poursuivi pour l'instruction suivante dans le cas d'une vérification positive, et, le processus de vérification échouant et le fragment de programme étant rejeté sinon.

20 7. Procédé selon l'une des revendications 3 à 6, caractérisé en ce que lorsque ladite instruction courante est la cible d'un appel de sous-routine, ledit processus de vérification vérifie que l'instruction précédente constitue un branchement inconditionnel, un retour de sous-routine ou une levée d'exception, ledit processus de
25 vérification, en cas de vérification positive, procédant à une réactualisation de la pile des types de variables par une entité de type retaddr, adresse de retour de la sous-routine, et, le processus de vérification échouant et le
30 fragment de programme étant rejeté sinon.

8. Procédé selon l'une des revendications 3 à 7, caractérisé en ce que lorsque l'instruction courante est la cible d'un gestionnaire d'exceptions, ledit processus de vérification vérifie que l'instruction précédente
 5 constitue un branchement inconditionnel, un retour de sous-routine ou une levée d'exception, ledit processus de vérification, en cas de vérification positive, procédant à une réactualisation de la pile des types par une entrée du type des exceptions, et, le processus de vérification
 10 échouant et le fragment de programme étant rejeté sinon.

9. Procédé selon l'une des revendications 3 à 8, caractérisé en ce que lorsque l'instruction courante est la cible d'une pluralité de branchements incompatibles, le processus de vérification échoue et le fragment de
 15 programme est rejeté.

10. Procédé selon l'une des revendications 3 à 9, caractérisé en ce que lorsque l'instruction courante n'est la cible d'aucun branchement, le processus de vérification continue par passage à une réactualisation de la pile des
 20 types.

11. Procédé selon l'une des revendications 3 à 10, caractérisé en ce que l'étape de vérification de l'effet de l'instruction courante sur la pile des types comporte au moins :

25 - une étape de vérification que la pile d'exécution des types contient au moins autant d'entrées que l'instruction courante comporte d'opérandes ;

~~- une étape de dépilement et de vérification que les~~
 30 types des entrées au sommet de la pile sont sous-types des types des opérandes de cette instruction ;

- une étape de vérification de l'existence d'un espace mémoire suffisant sur la pile des types pour procéder à l'empilement des résultats de l'instruction courante ;
- une étape d'empilement sur la pile des types de données attribués à ces résultats.

12. Procédé selon la revendication 11, caractérisé en ce que lorsque l'instruction courante est une instruction de lecture d'un registre d'adresse n, le processus de vérification consiste :

- à vérifier le type de donnée du résultat de cette lecture par consultation de l'entrée n du tableau des types de registres ;
- à déterminer l'effet de l'instruction courante sur la pile des types par dépilement des entrées de la pile correspondant aux opérandes de cette instruction courante et par empilement du type de données de ce résultat.

13. Procédé selon la revendication 11, caractérisé en ce que lorsque l'instruction courante est une instruction d'écriture d'un registre d'adresse m, le processus de vérification consiste :

- à déterminer l'effet de l'instruction courante sur la pile des types et le type t de l'opérande écrit dans ce registre d'adresse m ;
- à remplacer l'entrée de type du tableau des types de registres à l'adresse m par le type immédiatement supérieur au type précédemment stocké et au type t de l'opérande écrit dans ce registre d'adresse m.

14. Procédé de transformation d'un code objet d'un fragment de programme, dans lequel les opérandes de chaque instruction appartiennent aux types de données manipulées

par cette instruction, la pile d'exécution ne présente pas de phénomène de débordement, pour chaque instruction de branchement le type des variables de pile au niveau de ce branchement est le même qu'au niveau des cibles de ce branchement, en un code objet normalisé pour ce même fragment de programme, dans lequel les opérandes de chaque instruction appartiennent aux types de données manipulées par cette instruction, la pile d'exécution ne présente pas de phénomène de débordement, la pile d'exécution est vide à chaque instruction de branchement et à chaque instruction de cible de branchement, caractérisé en ce que ce procédé consiste, pour l'ensemble des instructions dudit code objet :

- à annoter chaque instruction courante par le type de données de la pile avant et après l'exécution de cette instruction, les données d'annotation étant calculées au moyen d'une analyse du flot des données relatif à cette instruction ;
- à détecter au sein desdites instructions et de chaque instruction courante l'existence de branchements respectivement de cibles de branchements pour lesquels ladite pile d'exécution n'est pas vide, l'opération de détection étant conduite à partir des données d'annotation du type des variables de pile allouées à chaque instruction courante, et en présence d'une détection d'une pile d'exécution non vide,
- à insérer des instructions de transfert des variables de pile de part et d'autre de ces branchements respectivement de ces cibles de branchements afin de vider le contenu de la pile d'exécution dans des registres temporaires avant ce branchement et de

rétablir la pile d'exécution à partir desdits registres temporaires après ce branchement, et à n'insérer aucune instruction de transfert sinon, ce qui permet d'obtenir un code objet normalisé pour ce même fragment de programme, dans lequel la pile d'exécution est vide à chaque instruction de branchement et à chaque instruction de cible de branchement, en l'absence de modification de l'exécution dudit fragment de programme. ;

15. Procédé de transformation d'un code objet d'un fragment de programme, dans lequel les opérandes de chaque instructions appartiennent aux types de données manipulées par cette instruction, et un opérande de type déterminé écrit dans un registre par une instruction de ce code objet est relu depuis ce même registre par une autre instruction de ce code objet avec le même type de donnée déterminé, en un code objet normalisé pour ce même fragment de programme, dans lequel les opérandes de chaque instruction appartiennent aux types de données manipulées par cette instruction, un seul et même type de donnée étant alloué à un même registre dans tout ledit code objet normalisé, caractérisé en ce que ce procédé consiste, pour l'ensemble des instructions dudit code objet :

- à annoter chaque instruction courante par le type de donnée des registres avant et après l'exécution de cette instruction, les données d'annotation étant calculées au moyen d'une analyse du flot des données relatif à cette instruction ;

- à effectuer une réallocation des registres par détection des registres d'origine employés avec des types différents, division de ces registres d'origine

en registres normalisés distincts, un registre normalisé pour chaque type de donnée utilisé, et une réactualisation des instructions qui manipulent les opérandes qui font appel auxdits registres normalisés.

5 16. Procédé selon la revendication 14, caractérisé en ce que l'étape consistant à détecter au sein desdites instructions et de chaque instruction courante l'existence de branchements respectivement de cibles de branchement pour laquelle la pile d'exécution n'est pas vide consiste,
10 suite à la détection de chaque instruction de rang i correspondant :

- à associer à chaque instruction de rang i un ensemble de nouveaux registres, un nouveau registre étant associé à chaque variable de pile active au niveau de
15 cette instruction ;

- à examiner chaque instruction détectée de rang i et à discriminer l'existence d'une cible de branchement respectivement d'un branchement, et dans le cas où l'instruction de rang i est une cible de branchement et
20 que la pile d'exécution au niveau de cette instruction n'est pas vide,

• pour toute instruction précédente, de rang i-1, constituée par un branchement, une levée d'exception ou un retour de programme, l'instruction détectée de
25 rang i n'étant accessible que par un branchement,

•• à insérer un ensemble d'instructions de chargement load à partir de l'ensemble de nouveaux registres antérieurement à ladite instruction détectée de rang i, avec redirection de tous les branchements vers
30 l'instruction détectée de rang i vers la première instruction de chargement load insérée ; et

- 5 • pour toute instruction précédente, de rang $i-1$,
continuant en séquence, l'instruction détectée de
rang i étant accessible à la fois par
l'intermédiaire d'un branchement et de l'instruction
précédente de rang $i-1$,
- 10 •• à insérer un ensemble d'instructions de sauvegarde
store vers l'ensemble de nouveaux registres
antérieurement à ladite instruction détectée de rang
 i et un ensemble d'instructions de chargement load à
partir de cet ensemble de nouveaux registres, avec
redirection de tous les branchements vers
l'instruction détectée de rang i vers la première
instruction de chargement load insérée, et, dans le
cas où ladite instruction détectée de rang i est un
15 branchement vers une instruction déterminée,
- pour toute instruction détectée de rang i constituée
par un branchement inconditionnel,
- 20 •• à insérer antérieurement à l'instruction détectée de
rang i une pluralité d'instructions de sauvegarde
store, à chaque nouveau registre étant associée une
instruction de sauvegarde ; et
- 25 • pour toute instruction détectée de rang i constituée
par un branchement conditionnel et pour un nombre
 $m > 0$ d'opérandes manipulés par cette instruction de
branchement conditionnel,
- 30 •• à insérer antérieurement à cette instruction
détectée de rang i une instruction de permutation,
swap-x, au sommet de la pile d'exécution des m
opérandes de l'instruction détectée de rang i et des
 n valeurs suivantes, cette opération de permutation

permettant de ramener au sommet de la pile d'exécution les n valeurs à sauvegarder dans l'ensemble des nouveaux registres, et

- 5 •• à insérer antérieurement à l'instruction de rang i un ensemble d'instructions de sauvegarde store vers l'ensemble de nouveaux registres, et
- à insérer postérieurement à l'instruction détectée de rang i un ensemble d'instructions de chargement load à partir de l'ensemble des nouveaux registres.

10 17. Procédé selon la revendication 15, caractérisé en ce que l'étape consistant à effectuer une réallocation des registres par détection des registres d'origine employés avec des types différents consiste :

- 15 - à déterminer les intervalles de durée de vie de chaque registre ;
- à déterminer le type de données principal de chaque intervalle de durée de vie, le type de données principal d'un intervalle de durée de vie j pour un

- 20 registre r étant défini par la borne supérieure des types de données stockées dans ce registre r par les instructions de sauvegarde store appartenant à l'intervalle de durée de vie j ;
- à établir un graphe d'interférences entre les intervalles de durée de vie, ce graphe d'interférences
- 25 consistant en un graphe non orienté dont chaque sommet est constitué par un intervalle de durée de vie et dont les arcs entre deux sommets j_1 et j_2 existent si un

- sommet contient une instruction de sauvegarde adressée au registre de l'autre sommet ou réciproquement ;
- 30 - à traduire l'unicité d'un type de donnée alloué à chaque registre dans le graphe d'interférences en

ajoutant des arcs entre toute paire de sommets du graphe d'interférences tant que deux sommets d'une paire de sommets n'ont pas le même type de donnée principal associé ;

- 5 - à effectuer une instanciation du graphe d'interférences, par attribution à chaque intervalle de durée de vie d'un numéro de registre de telle manière qu'à deux intervalles de vie adjacents dans le graphe d'interférences soient attribués des numéros de registres différents.
- 10

18. Système embarqué reprogrammable par téléchargement de fragments de programmes, comportant au moins un microprocesseur, une mémoire vive, un module d'entrées/sorties, une mémoire non volatile reprogrammable
15 électriquement et une mémoire permanente dans laquelle sont implantés un programme principal et une machine virtuelle permettant l'exécution du programme principal et d'au moins un fragment de programme par l'intermédiaire dudit microprocesseur, caractérisé en ce que ledit système

20 embarqué comporte au moins un module de programme de gestion et de vérification d'un fragment de programme téléchargé suivant le protocole de gestion d'un fragment de programme téléchargé selon l'une des revendications 1 ou 2, ledit module de programme de gestion et de
25 vérification étant implanté en mémoire permanente.

19. Système embarqué selon la revendication 18, caractérisé en ce que celui-ci comporte au moins un module
~~de sous-programme de vérification d'un fragment de~~
programme téléchargé, suivant le processus de vérification
30 selon l'une des revendications 3 à 13.

20. Système de transformation d'un code objet d'un fragment de programme, dans lequel les opérandes de chaque instruction appartiennent aux types de données manipulées par cette instruction, la pile d'exécution ne présente pas de phénomène de débordement, pour chaque instruction de branchement le type de variables de pile au niveau de ce branchement est le même qu'au niveau des cibles de ce branchement et un opérande de type déterminé écrit dans un registre par une instruction de ce code objet est relu depuis ce même registre par une autre instruction de ce code objet avec le même type de donnée déterminé, en un code objet normalisé pour ce même fragment de programme dans lequel les opérandes de chaque instruction appartiennent aux types de données manipulées par cette instruction, la pile d'exécution ne présente pas de phénomène de débordement, la pile d'exécution est vide à chaque instruction de branchement et à chaque instruction de cible de branchement, un seul et même type de donnée étant alloué à un même registre dans tout ledit code objet normalisé, caractérisé en ce que ledit système de transformation comporte au moins, implanté en mémoire de travail d'un ordinateur de développement ou d'une station de travail, un module de programme de transformation de ce code objet en un code objet normalisé suivant le procédé selon les revendications 14 et 15, ce qui permet d'engendrer un code objet normalisé pour ledit fragment de programme satisfaisant aux critères de vérification de ce fragment de programme téléchargé.

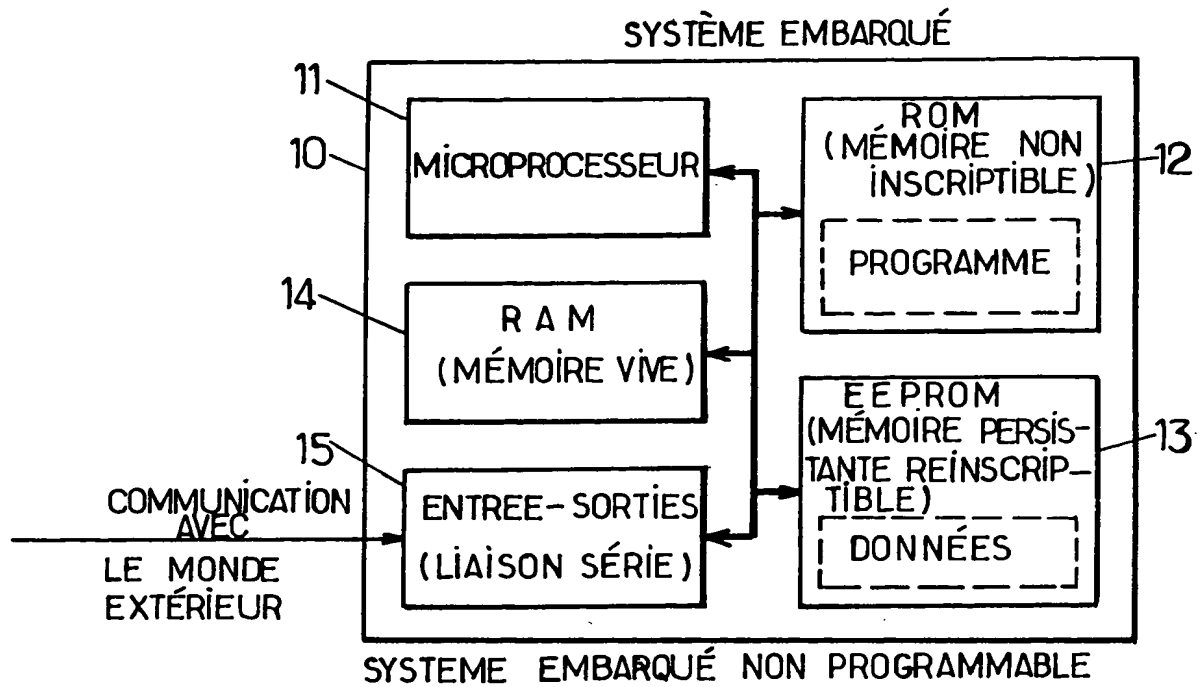
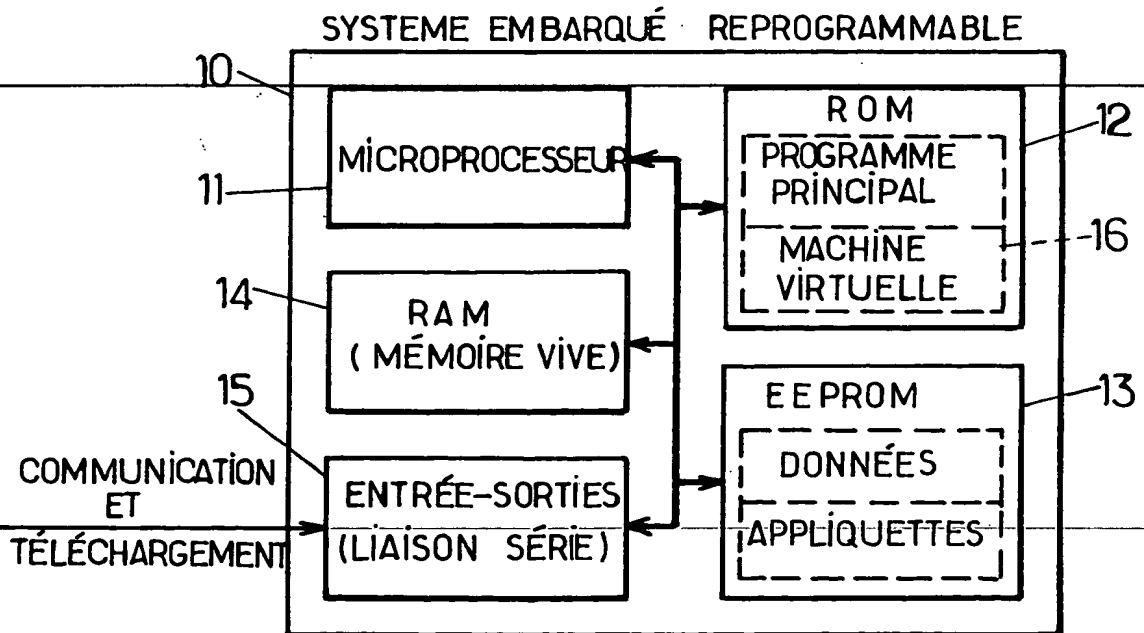


FIG.1a. (ART ANTÉRIEUR)



SYSTEME EMBARQUÉ REPROGRAMMABLE PAR TÉLÉCHARGEMENT D'APPLIQUETTES

FIG.1b. (ART ANTÉRIEUR)

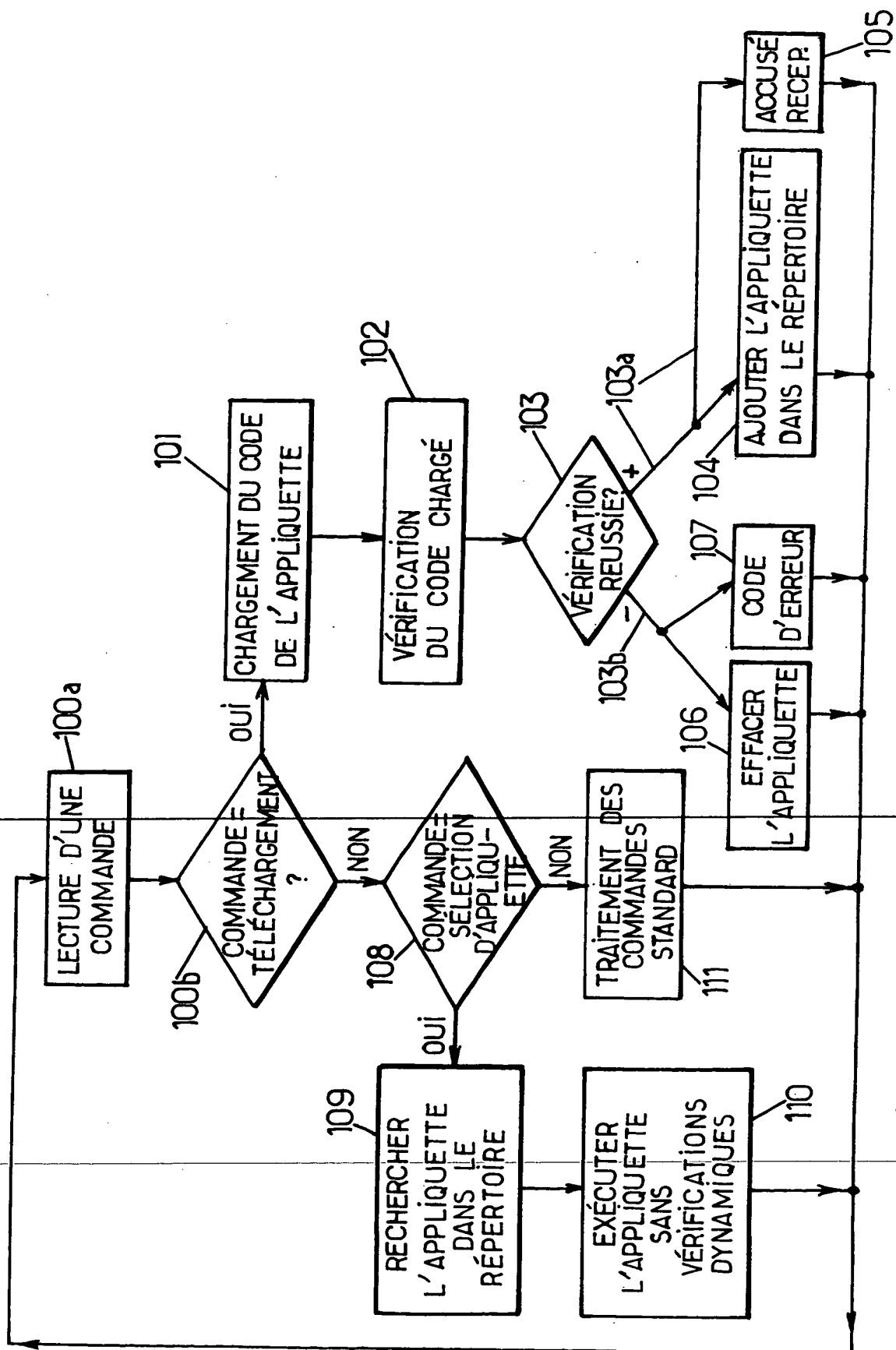


FIG.2. PROTOCOLE DE GESTION

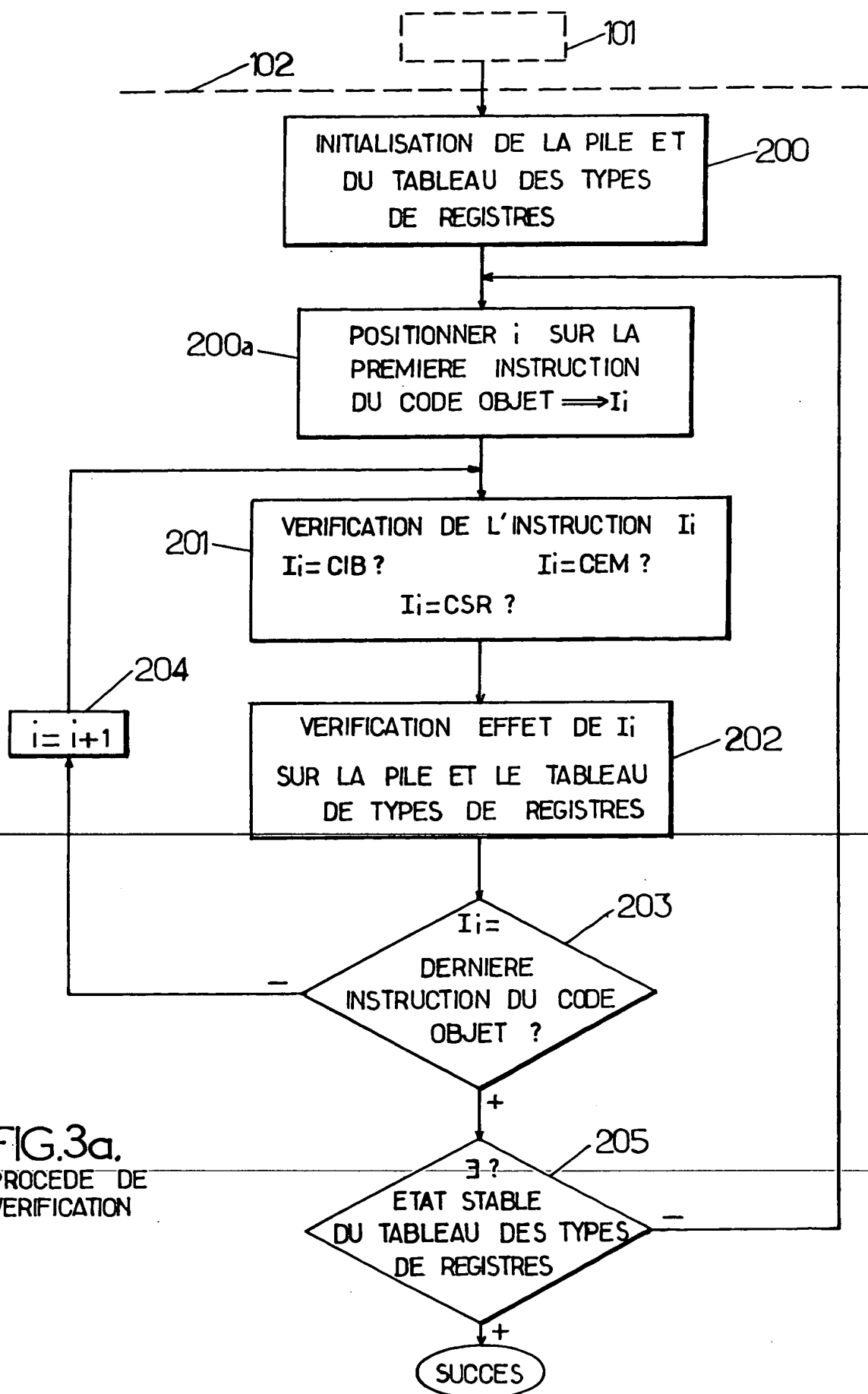


FIG. 3a.
PROCEDURE DE
VERIFICATION

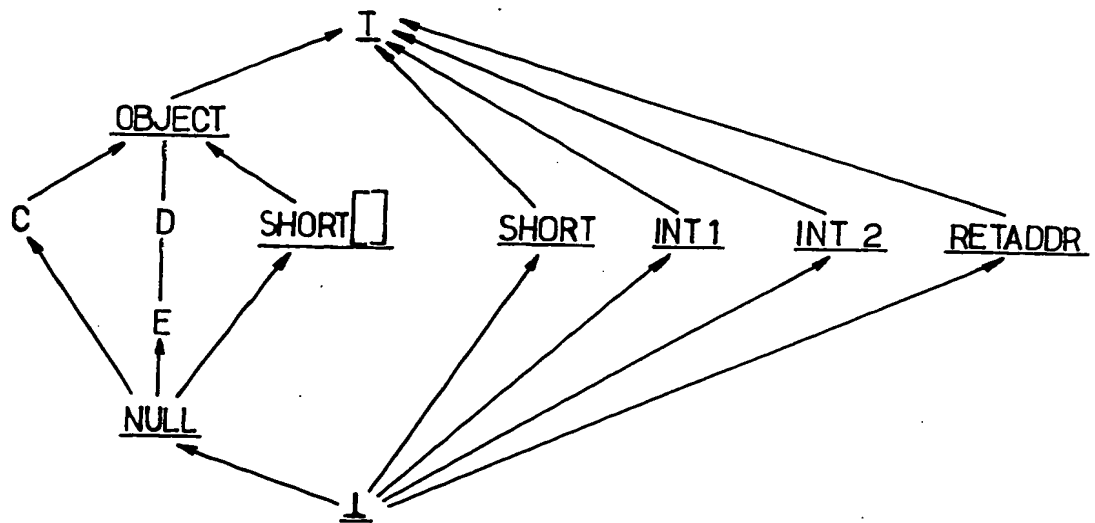


FIG.3b. TYPES DE DONNEES ET RELATION DE SOUS-TYPAGE

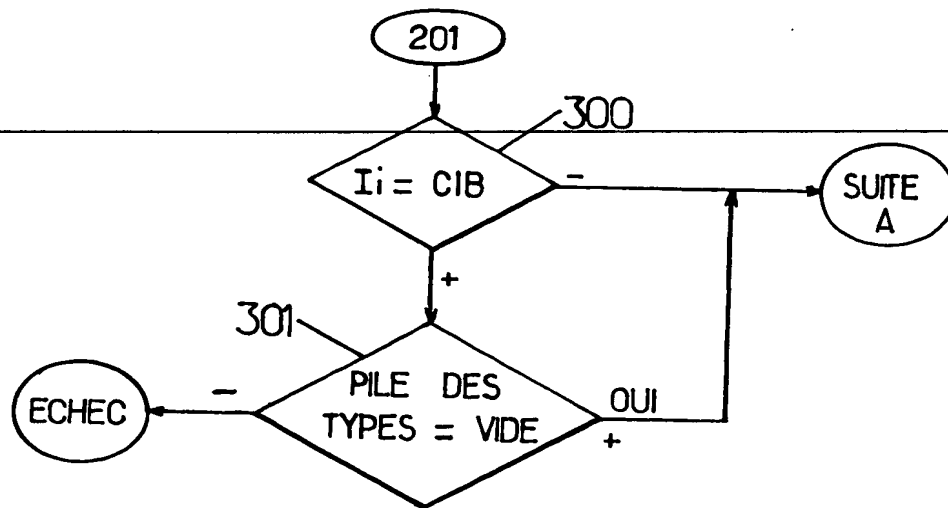


FIG.3c. VERIFICATION: GESTION D'UNE INSTRUCTION DE BRANCHEMENT

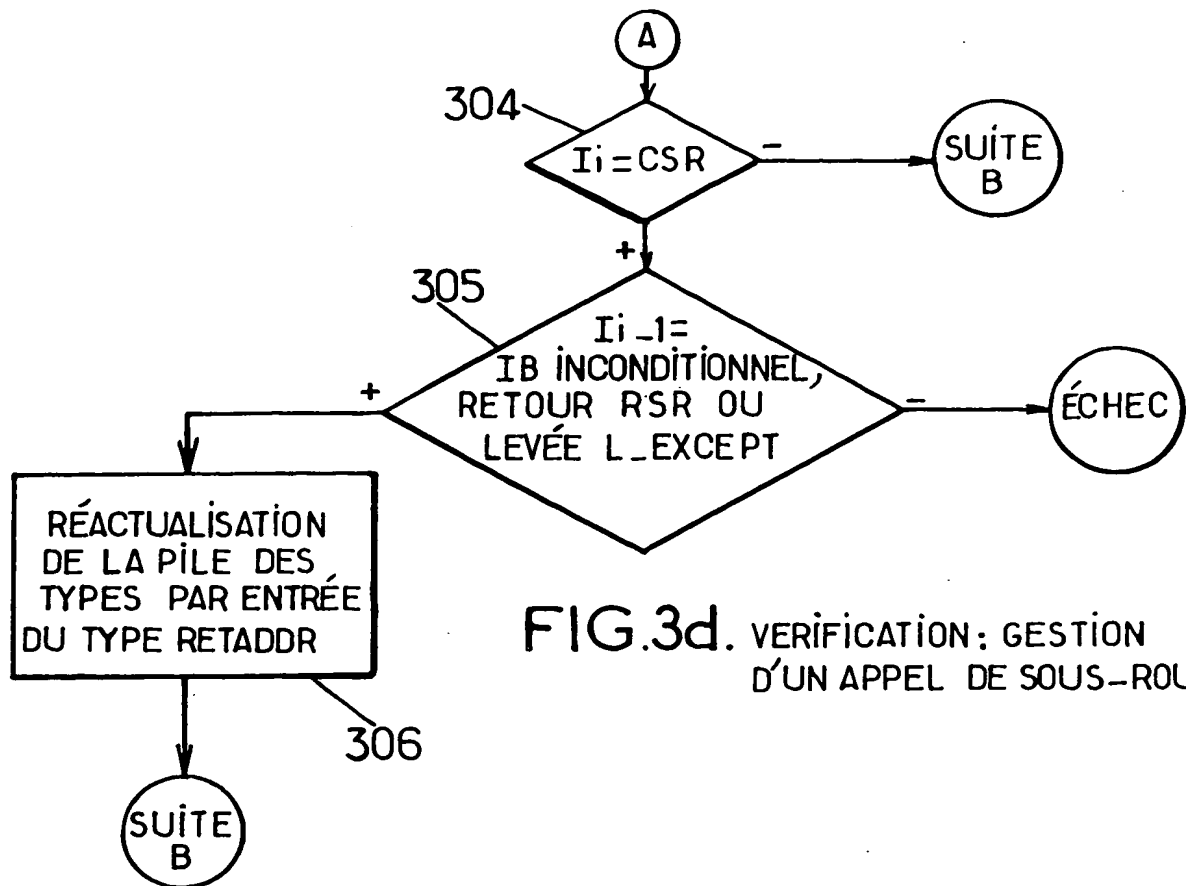


FIG.3d. VERIFICATION: GESTION
D'UN APPEL DE SOUS-ROUTINE

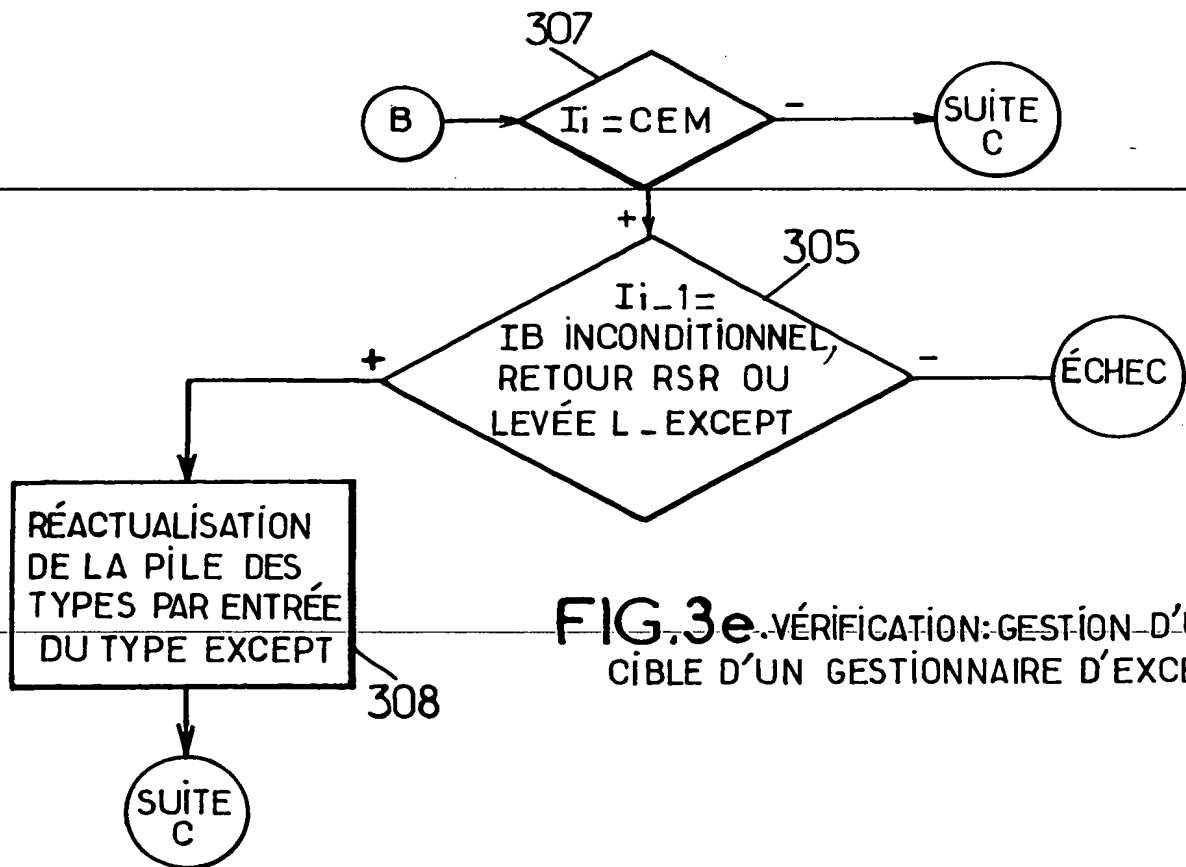


FIG.3e. VERIFICATION: GESTION D'UNE
CIBLE D'UN GESTIONNAIRE D'EXCEPTION

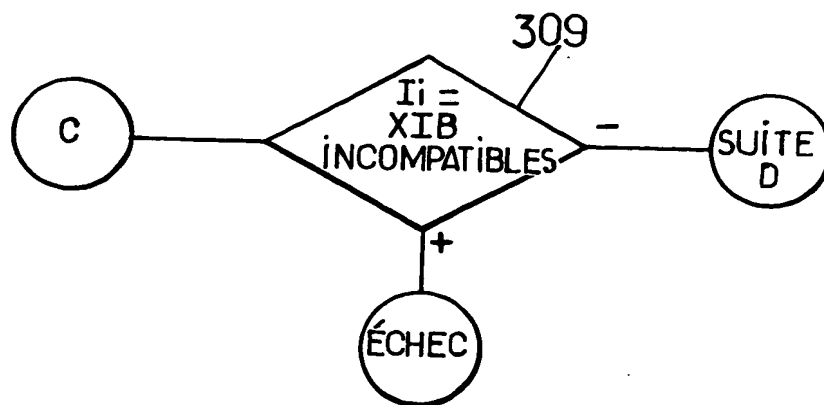


FIG. 3f. VÉRIFICATION: GESTION DE LA CIBLE DE BRANCHEMENTS INCOMPATIBLES

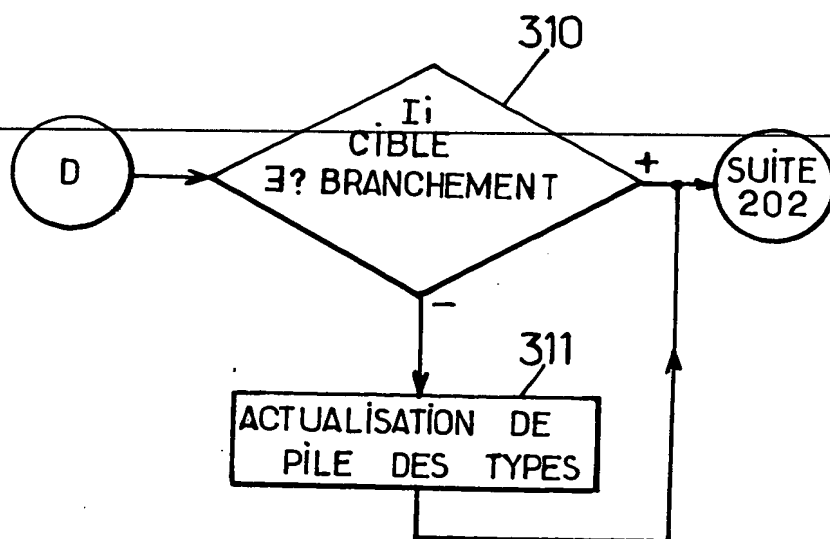


FIG. 3g. VÉRIFICATION: GESTION DE L'ABSENCE DE CIBLE DE BRANCHEMENT

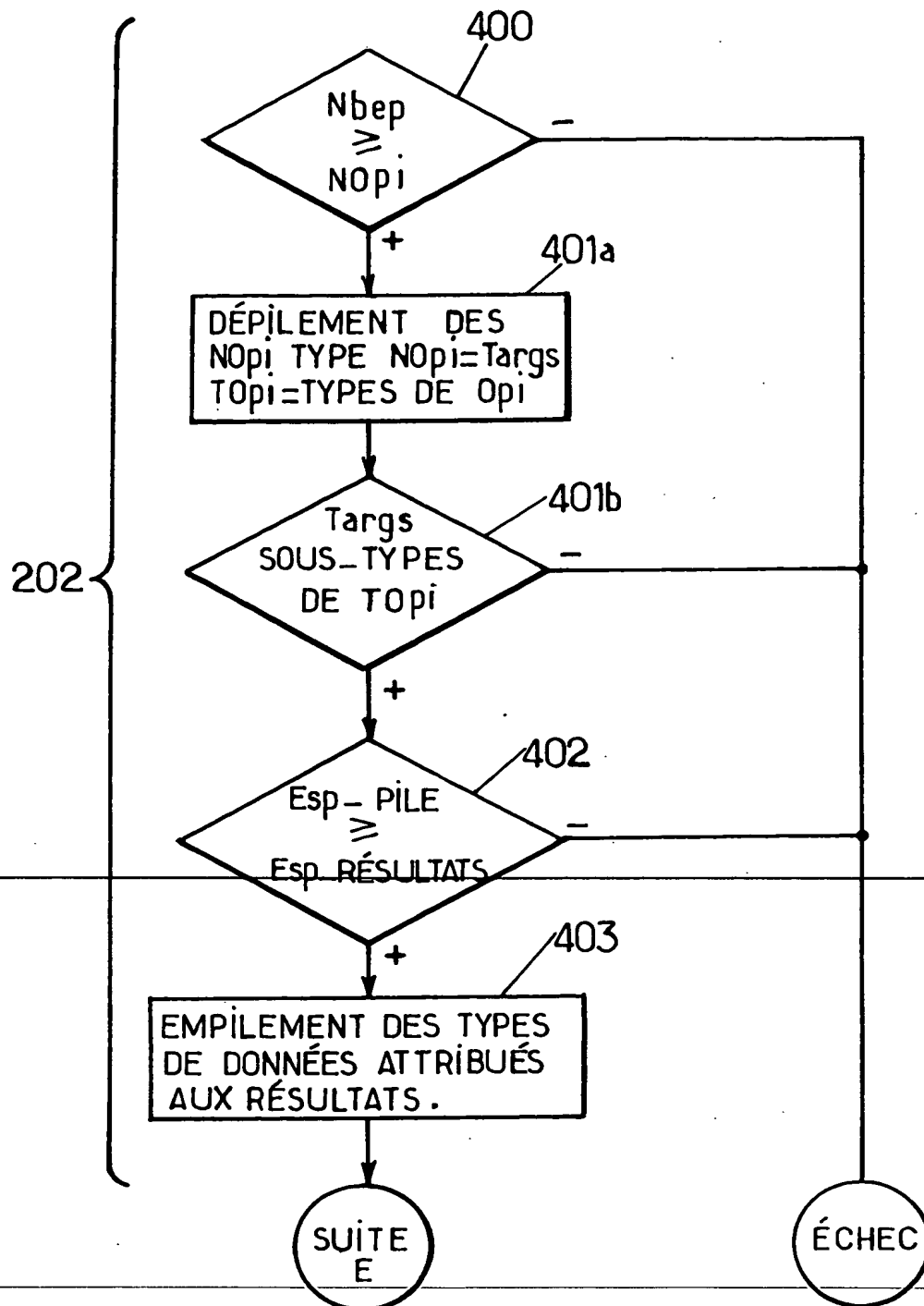


FIG.3h. VÉRIFICATION: EFFET DE L'INSTRUCTION COURANTE
SUR LA PILE DES TYPES

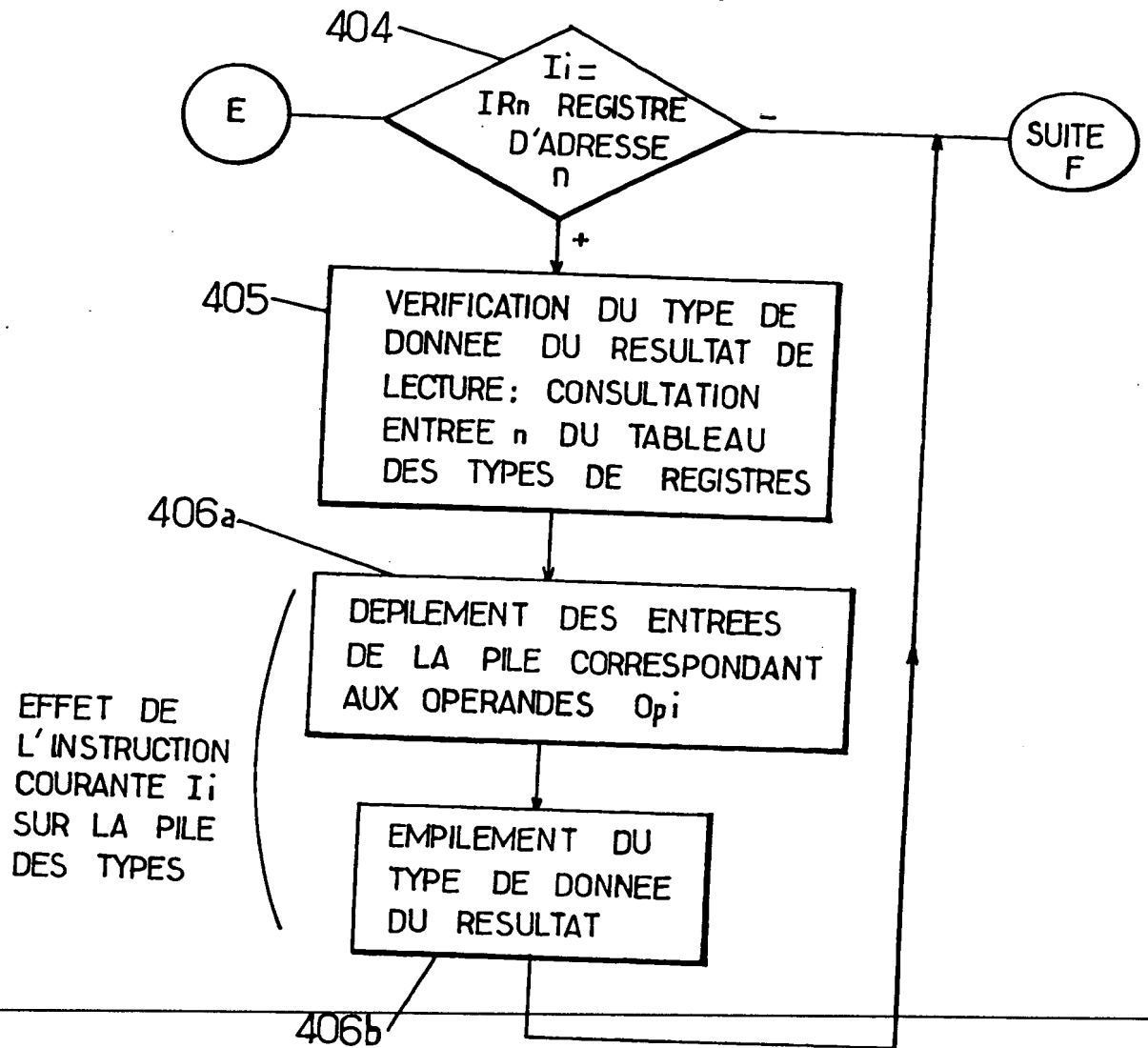


FIG.3i. VERIFICATION: GESTION D'UNE INSTRUCTION DE LECTURE D'UN REGISTRE

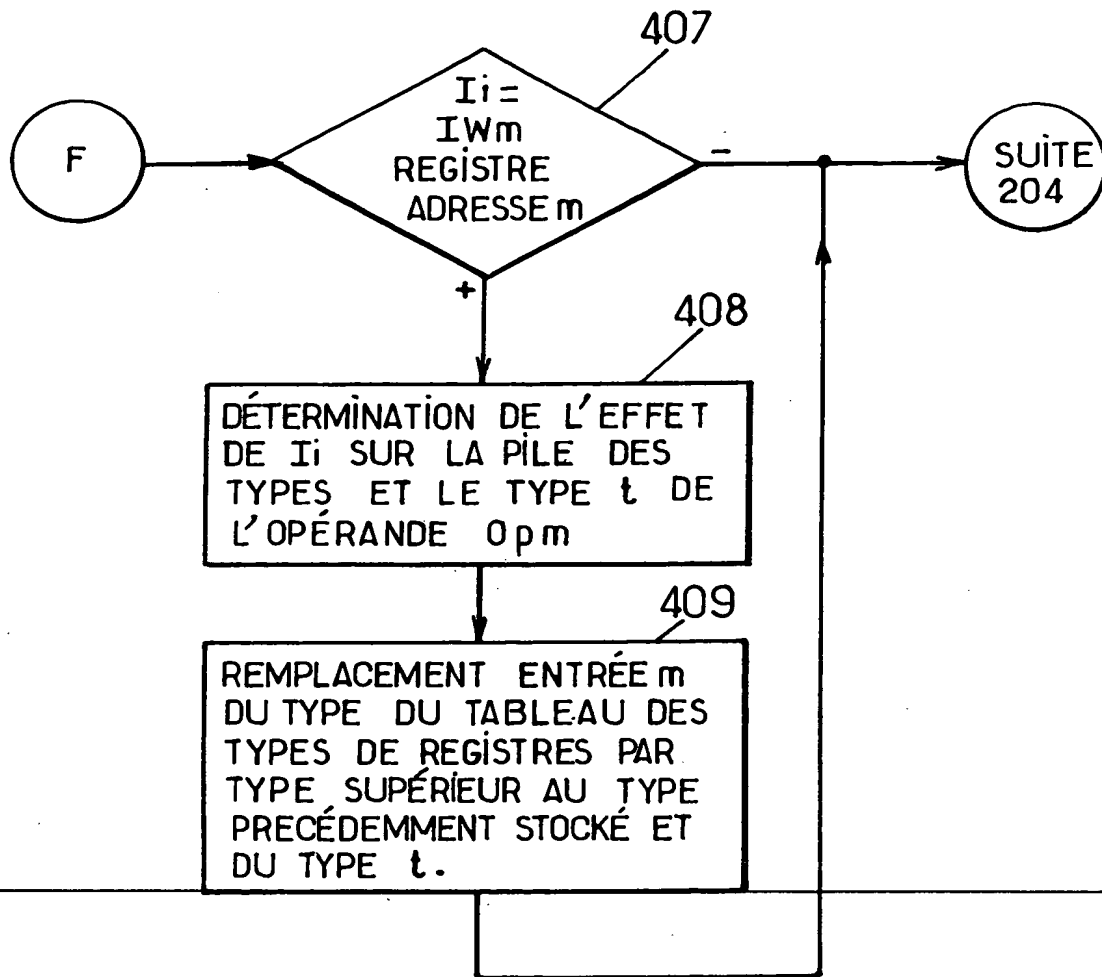


FIG.3j. VÉRIFICATION: GESTION D'UNE INSTRUCTION
D'ÉCRITURE D'UN REGISTRE

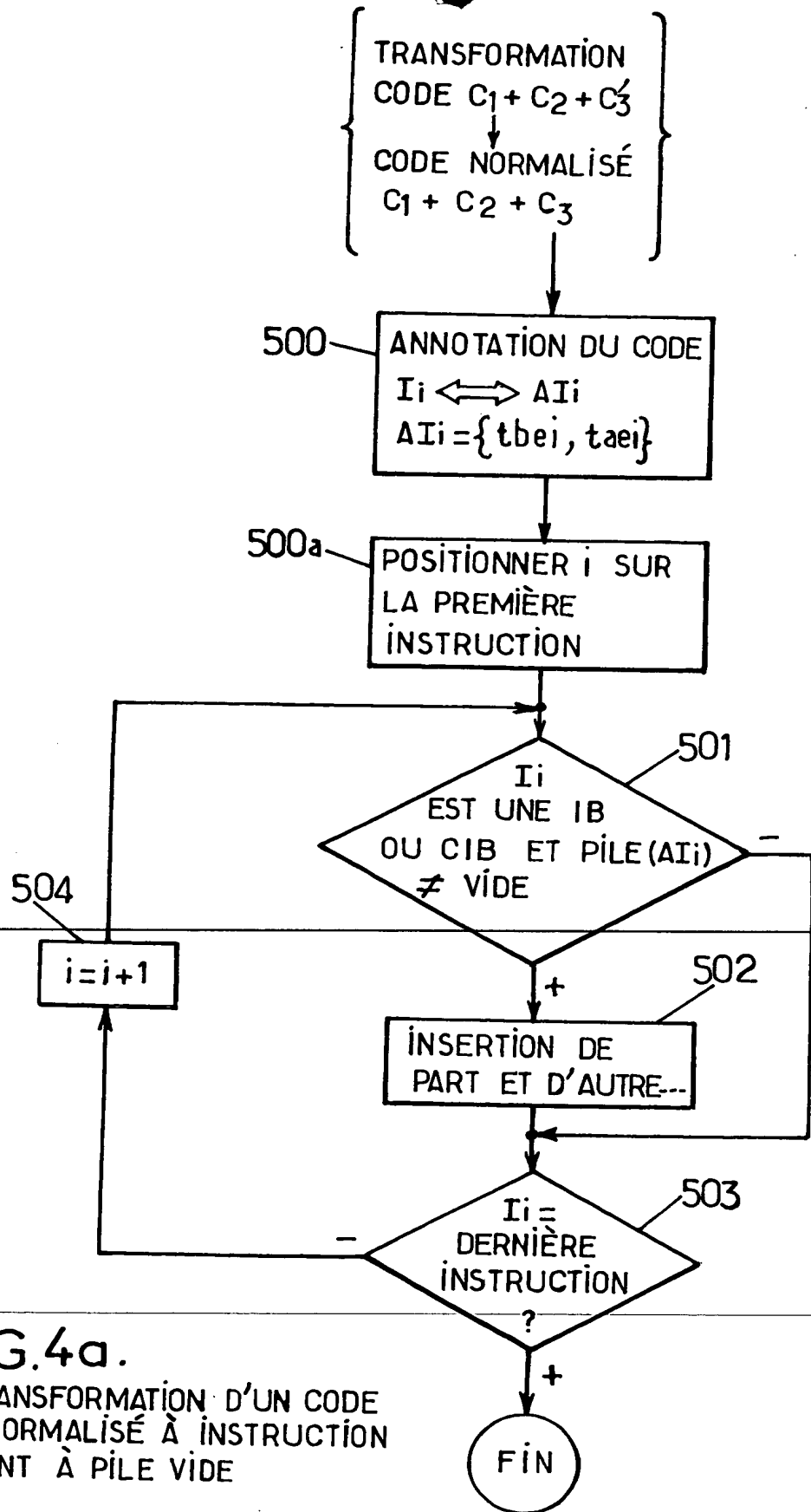


FIG.4a.

PROCÉDÉ DE TRANSFORMATION D'UN CODE
EN UN CODE NORMALISÉ À INSTRUCTION
DE BRANCHEMENT À PILE VIDE

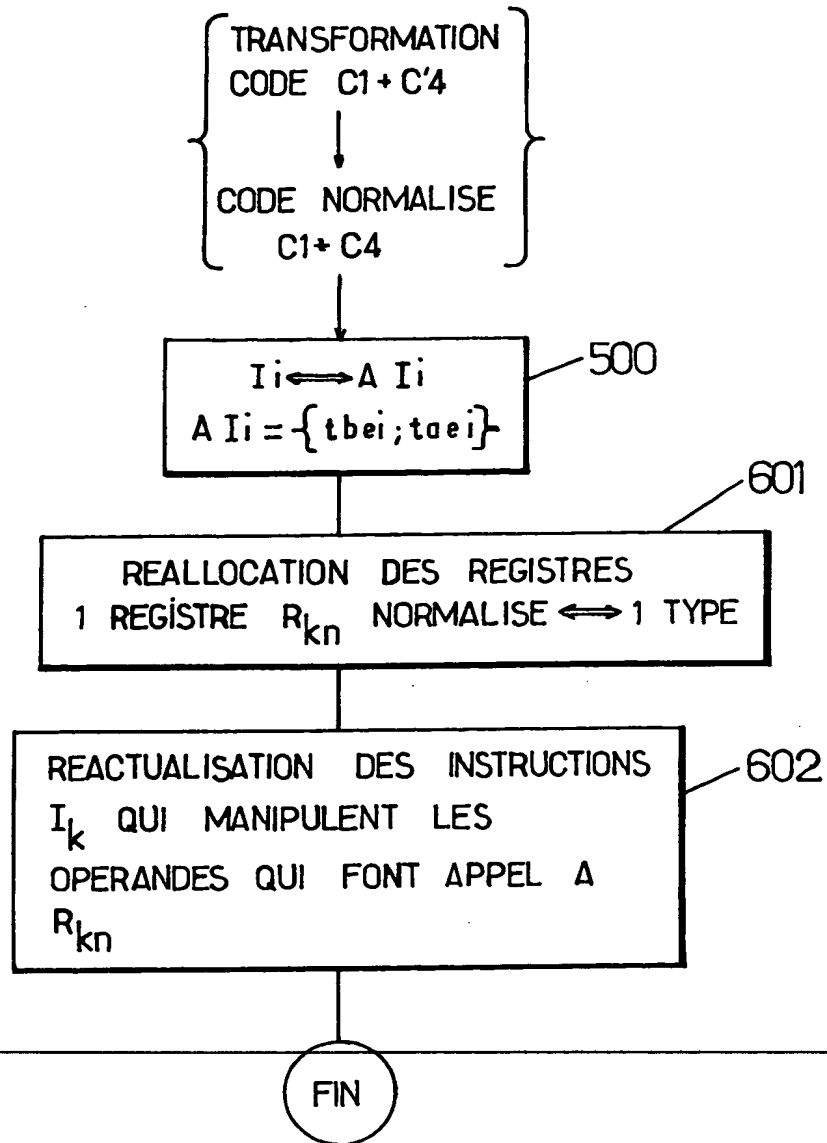


FIG.4b. PROCEDE DE TRANSFORMATION D'UN CODE
EN UN CODE NORMALISE FAISANT APPEL
A DES REGISTRES TYPE

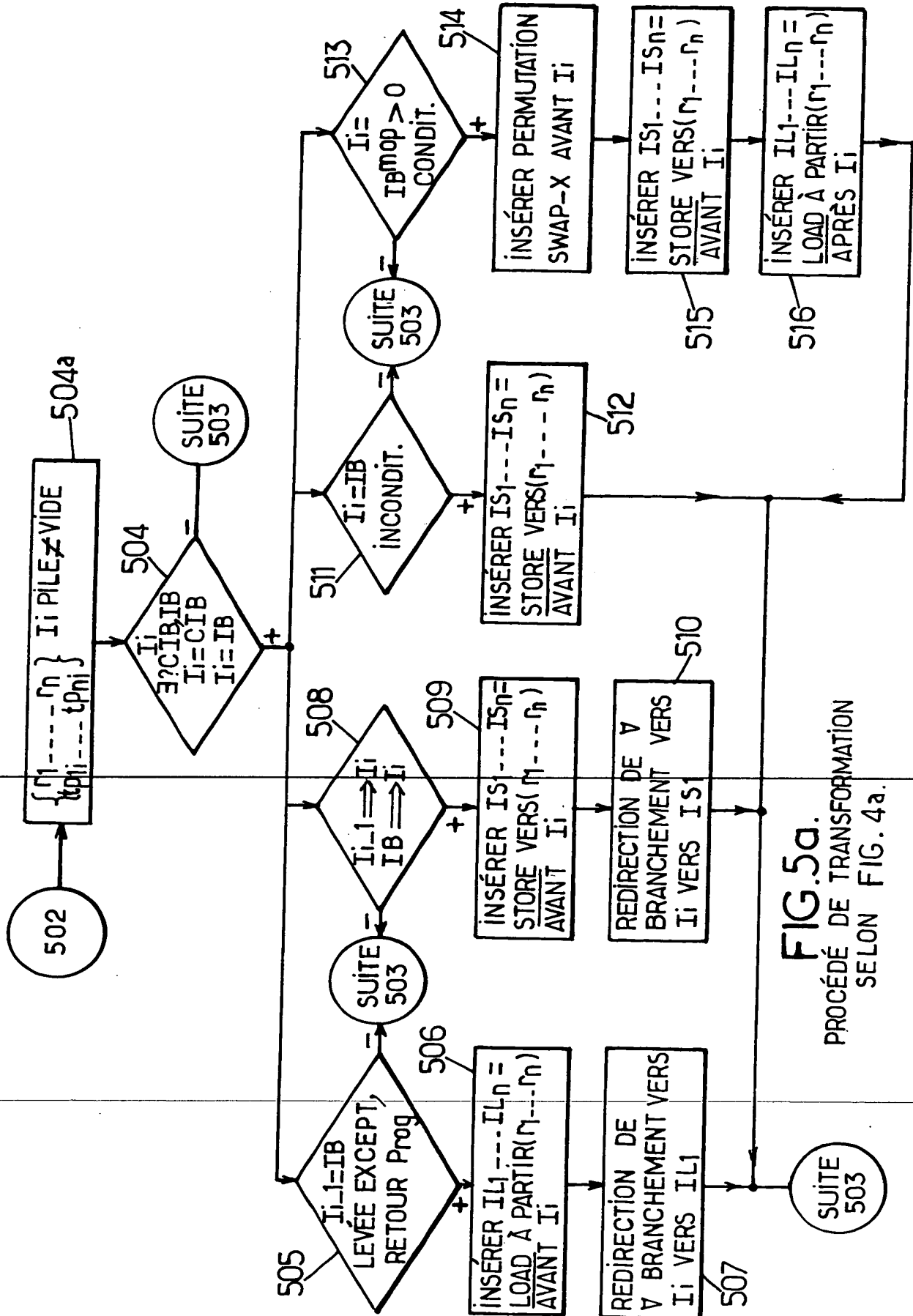


FIG. 5a.
PROCÉDÉ DE TRANSFORMATION
SELON FIG. 4a.

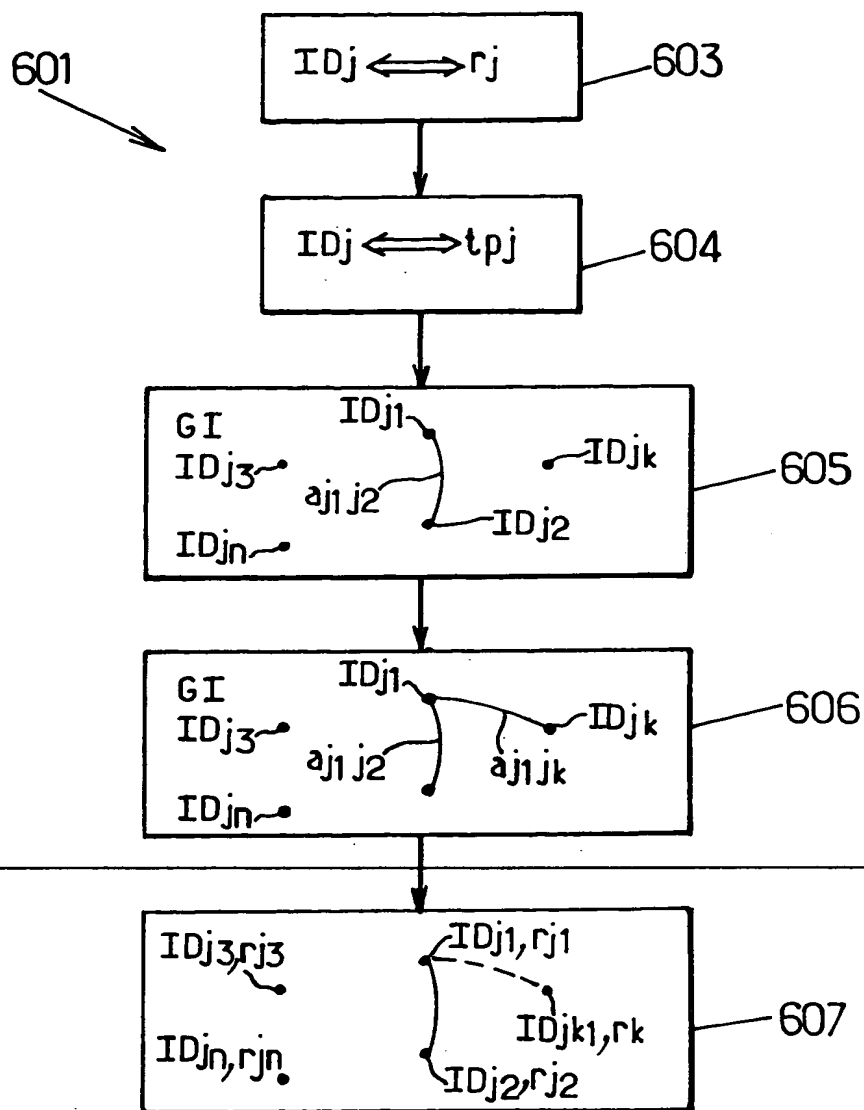


FIG. 5b.
 PROCÉDÉ DE TRANSFORMATION
 SELON LA FIG. 4b.

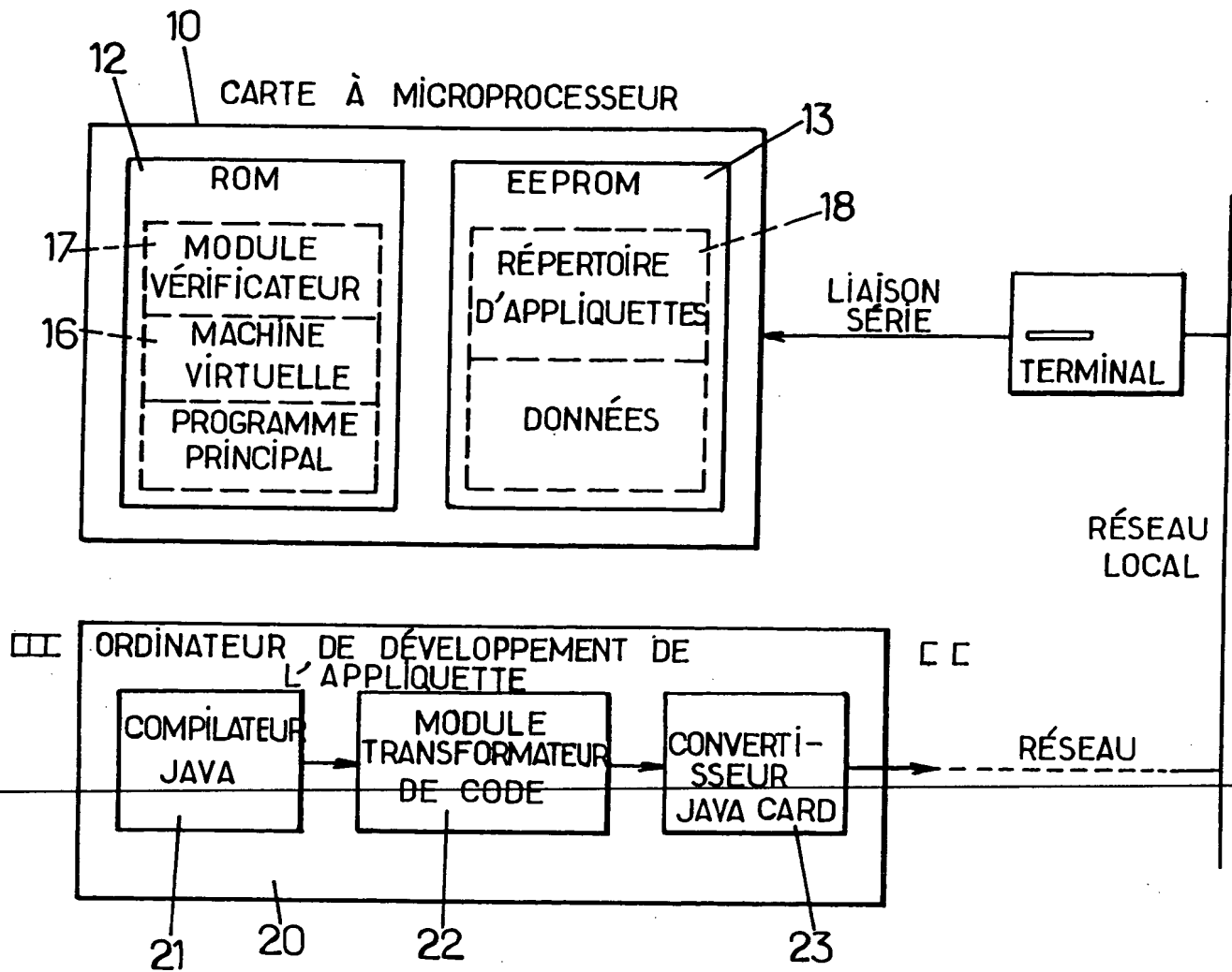


FIG. 6.